A Distributed Publish/Subscribe Notification Service for Pervasive Environments

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades
eines Doktor-Ingenieurs (Dr.-Ing.)

von Diplom-Informatiker

Andreas Zeidler

aus Mülheim an der Ruhr

Referent: Prof. A. Buchmann, PhD, TU Darmstadt
Korreferent: Prof. Dr. K. Geihs, TU Berlin
Tag der Einreichung: 01.09.2004
Tag der mündlichen Prüfung: 04.11.2004

Darmstädter Dissertationen D17

Summary

Based on the success of mobile telephony in the recent past, many observers expect mobility in conjunction with mobile devices to open up a wide field for novel applications. Many experts predict the arrival of new services, such as mobile commerce, location-based services, multimedia messaging, and mobile gaming. They claim that this new class of *mobile* applications will constitute a main driving-force for technological advancements at least for the next decade. Also, in the third generation of mobile telephony, we expect to observe the logical next step: the convergence of mobile appliances into a new generation of *smart* devices, such as smartphones. Today, we can distinguish three basic categories of mobile devices: (i) specialized devices, such as special-purpose systems or mobile phones, (ii) devices that are more flexible, such as *personal digital assistants* (PDAs), and (iii) real general-purpose devices, such as laptop computers. The next generation of devices will be unifying distinct features of the categories named above. Such devices will become flexible, lightweight, and mobile at the same time. This meets a basic requirement found in the vision of *ubiquitous* and *pervasive computing*. The latter vision places its main focus on smart mobile devices as the enabling technology for interaction of mobile users with the surrounding infrastructure.

Another trend expected is based on the observation that more and more artifacts in the infrastructure will be equipped with processors and—more importantly—networking interfaces. Therefore, they are able to generate as well as receive data. The original thesis of ubiquitous computing expects a mobile user to be embedded into surroundings filled with communicating and interacting artefacts, all serving the spontaneous needs of the user. Moreover, we are convinced that the interaction between users and the surroundings in highly mobile and dynamic settings has to be mediated by a common middleware platform, together with personalized devices and specialized services, facilitating the needs of mobile users.

This basic system model of nomadic users and smart infrastructures poses a number of challenges for such middleware support. First of all, mobility by itself requires different paradigms for interaction than those found in classical distributed systems. Many paradigms, well-established in static distributed systems are likely to fail when applied to these new settings. One prominent example among many is the *request/reply* paradigm, which is too static and tight-coupled to be successful in dynamic mobile settings. Here, different paradigms, like loose-coupling and data-centric computing, are more likely to succeed.

The next key challenge for middleware is to support mobile applications to react “smartly” to changes of their execution environment. Users of such applications obviously expect their electronic helpers to adapt themselves to the current situation they are used in. A well-known example is to turn off the ringer tones of a mobile phone when the user is in a meeting situation. Such adaptation is part of what usually is called context- or situation-aware computing. The challenge for middleware support lies here in providing means to retrieve context information from the environment on a syntactic and semantic level. Here we face issues of heterogeneity, together with efficient filtering of large volumes of information available.

Another rationale—as well as challenge—for middleware support in dynamic and mobile scenarios is the need to decouple producers and consumers of data in the system in *time* and *space*. For the systems considered in this thesis, often it is not feasible for producers and consumers to "know" each other, especially due to the number of participants or resource constraints. Effective means for anonymous interaction are therefore essential. Moreover, for mobile clients the receiver cannot be assumed to be online at the same time the sender produces the data. Again, a middleware solution can provide facilities for buffering and access to past information.

The scale of pervasive systems we envision is also a challenge. On the one hand, systems will grow in physical size, like spanning a whole city. On the other hand, systems also can be rather small in size, but dense in the number of processors and applications contained within. Thus, the key challenge is to provide a communication infrastructure in which data and information is still manageable even for small devices while communication remains efficient and scalable.

Altogether, we are convinced that this constitutes a strong demand for a mediator *between* producers and consumers of data, i.e., a middleware solution.

This thesis presents solutions to the challenges listed above using mechanisms that are based on a distributed publish/subscribe notification service. The main contributions are:

Requirement analysis. As a basis for assessing the novelty of pervasive computing environments in comparison to conventional distributed systems, we provide a thourough analysis of the problem domain. The main result is a taxonomy and a number of requirements on which we built and assess our own solutions. Among the requirements, the need for proper support for mobility and environment-awareness is of outstanding importance. Moreover, we compare several different communication paradigms for distributed systems to identify one which will serve best as the basis for extensions needed in pervasive systems. We identify the well-established publish/subscribe paradigm as a suitable basis for such extensions. It already addresses a number of requirements, therefore forming a sound foundation, but falls short to meet others as detailed below.

Mobility support. As a first step towards a content-based publish/subscribe notification infrastructure for pervasive systems, we introduce an important mechanism for transparency of mobility. This is a common requirement for clients of the infrastructure that roam freely. Certain aspects of the handling of this issue are located in the infrastructure and are opaque to the client. This can be beneficial for a client either because it is not aware of its own mobility, e.g., together with legacy applications, or deliberately wants to delegate some aspects into the infrastructure. Therefore, we devised a relocation algorithm that facilitates location transparency, offering the possibility to transfer existent event-based applications seamlessly into mobile environments. The algorithm extends the existing content-based routing infrastructure to support non-interrupted, sender-FIFO ordered delivery of notifications in the mobile case, without having a client even to be aware of this extension.

Location-dependent subscriptions and notifications. The next logical step is to provide means for mobility-aware applications to express their interest in events and data related to their current environment. To do so, we choose *location* as a well-understood and rich *indexing scheme* on such information. First, most information can be related to some location and next, we need strong selection criteria to distinguish relevant from irrelevant information. However, to make *location* usable together with a content-based publish/subscribe notification service, we introduced a special location model. It serves as the foundation for location-dependent subscriptions and notifications, respectively. The challenge from the point of view of the publish/subscribe infrastructure is twofold: first, hiding the details and burdens of adaptation of location-dependent subscriptions to the current position of a client. Second, due to the uncertainty of the client position and movement, to keep delivery of information timely and accurate and to keep the network load for the client bearable. We

introduce an adaptive algorithmic solution that addresses both challenges. It offers delivery guarantees, normally only found together with network flooding, but contrary to flooding, leverages the information about the client subscription's relevancy in space to restrict the degree to which uncertainty of location related message delivery is necessary. Thereby, an effective means to express and implement location-awareness is introduced.

Decoupling in space and time. To a large degree the previous solutions, together with the basic publish/subscribe paradigm, already decouple sender and receipient of data in space and time. However, an inherent danger of asynchronous, anonymous communication is the unpredictability of when data is generated. This can be harmful in cases where a mobile client needs a certain number of notifications to reach a consistent state, from which its execution can commence. Here, we propose techniques to access past information. We devise mechanisms in the infrastructure, enabling a client to minimize the time-span it has to listen for new notifications. This can be done by virtually relocating the arrival time of a client at a new location into the past. Hence, we establish distributed buffers in the infrastructure together with a set of search and consolidation strategies, tailored to minimize the bootstrapping latency experienced by a client. Together with the algorithms introduced above, a considerable decoupling of producers and consumers in space *and* time is achieved.

A framework for the development of context-aware applications. We identify *context* to be an important input for applications in pervasive computing systems. Usually, such context data is the result of changes in the volatile external computing environment the client operates in. Adaption therefore is reactive in nature. We analyze the impact on the development of context-sensitive applications and based on this, we introduce a framework for the structured development of such applications. Within this framework we show how, at designtime, semantically high-level context information can be decomposed stepwise to match the semantics and syntax of data found in a system at runtime. At runtime we leverage well-defined event operators, such as event composition and aggregation to generate the wanted input on the level of applications. Some aspects of the framework resemble mechanisms also found in the rather recent paradigm of *model driven development* (MDD).

Summing up, this thesis provides solutions to the question of how the successfully deployed publish/subscribe paradigm can be extended to serve as a middleware platform for pervasive computing systems. It does so by proposing concrete architectures, algorithms, and frameworks, which additionally have been implemented prototypically.

Zusammenfassung

In den letzten Jahren konnte die Welt den beeindruckenden Erfolg der mobilen Telefonie miterleben. Das Mobiltelefon hat sich innerhalb weniger Jahre vom Exoten zum Massenprodukt gewandelt. Parallel dazu haben sich weitere mobile Geräte enorm fortentwickelt, wie *Persönliche Digitale Assistenten* (PDAs) und Notebook-Computer. Konsequenterweise beobachten wir heute den logischen nächsten Schritt. Mobiltelefone und PDAs konvergieren technologisch zu einer neuer Geräteklasse, die so genannten *Smartphones*. Sie zeichnen sich dadurch aus, dass sie ausreichende Rechenleistung mit der Möglichkeit der mobilen Kommunikation vereinen. Außerdem wird sich in Zukunft die Funktionalität dieser Geräte erweitern, insbesondere durch das Hinzufügen externer Sensoren, z.B. für das *Globale Positioning System* (GPS). Denkbar sind darüber hinaus weitere Sensoren zur Feststellung bestimmter Parameter der aktuellen Umgebung.

Daher wird erwartet, dass mobile Geräte und ihre Applikationen die größten technologischen Trends des nächsten Jahrzehnts sein werden, kommerziell ebenso wie in der Forschung. Manche gehen noch einen Schritt weiter und prognostizieren, dass komplementär zu den mobilen Geräten ihre *Umgebung* ebenfalls *smart* werden wird. Solche Umgebungs-Intelligenz äußert sich darin, dass die Infrastruktur selber angereichert ist mit allen erdenklichen Artefakten, welche über hinreichende Rechenleistung und (vor allem) eine Netzwerk-Schnittstelle zur Kommunikation verfügen. Diese Form der hochgradig vernetzen mobilen Systeme wurde erstmals von Mark Weiser postuliert und von diesem mit dem Begriff *Ubiquitous Computing* (Ubiquitäres Rechnen) [Wei93; Wei91] belegt. Nach seiner klassischen Definition findet ein großer Teil der eigentlichen Applikationen in der Infrastruktur statt. Im Extrem sogar ausschließlich dort. Das wird dann u.a. als „Ambient Computing" bezeichnet. Ein anderer Name für eine ähnliche Sicht auf mobile Systeme ist „Pervasive Computing," eingeführt von IBM [IBM01b]. Der Fokus dieses Begriffs ist leicht verschoben und ruht auf den notwendigen Technologien, die es dem Benutzer eines Systems ermöglichen, mit seiner (elektronischen) Umgebung mit Hilfe eines *personalisierten* Gerätes zu interagieren. Die grundlegende Annahme ist, dass ein Benutzer einem persönlichen Helfer grundsätzlich mehr vertraut als wechselnden Systemen wie sie in unterschiedlichsten Dienstnutzungsszenarien vorgefunden werden. Solche Dienstnutzungsszenarien reichen von einfachen Druckdiensten an Flughäfen oder Bahnhöfen bis zu lokationsbewussten Diensten, die abhängig von der aktuellen Position ihre Dienstleistung entsprechend anpassen („*Location-based Services*"). Die zentrale Herausforderung ist die Notwendigkeit der Anpassung an die physikalische ebenso wie die Ausführungsumgebung, in dem sich ein persönlicher elektronischer Helfer aktuell befindet. Eine Notwendigkeit zur flexiblen Anpassungsfähigkeit wird oftmals auch als Kontextbewusstsein bezeichnet. Benutzer mobiler Geräte, die mit der lokalen Umgebung interagieren, erwarten zu recht ein gewisses Maß an „smartem" Verhalten der mobilen Geräte. In unterschiedlichen Situationen sollen bestimmte Verhaltensweisen automatisch angepasst werden. Zum Beispiel sollte während einer Sitzung ein mobiles Telefon auf stummen Alarm geschaltet oder sogar ganz auf die Mailbox umgeleitet werden. Um ein solches Verhalten zu realisieren ist daher ein Kontextbewusstsein notwendig. Kontextbewusstsein kann nur dann existieren, wenn das mobile Gerät in der Lage ist, mit seiner (wechselnden) Umgebung zu kommunizieren und bestimmte

notwendige Informationen über den aktuellen Kontext einzuholen.

Aus der Vision des Ubiquitären Rechnens und der Vision der Welt, die durchdrungen ist von unzähligen Sensoren, Aktuatoren und anderen „intelligenten" Artefakten, mit denen personalisierte mobile Geräte interagieren müssen, sehen wir aus infrastruktureller Sicht eine Reihe fundamentaler Herausforderungen:

Mobile Systeme. Die erste und offensichlichste Herausforderung ist der hohe Grad an Mobilität in ubiquitären Dienstnutzungsszenarien. Durch die Mobilität der Benutzer sind auch deren digitalen Helfer mobil, was einen großen Einfluss auf den generellen Aufbau mobiler Gesamtsysteme hat. Als ein prominentes Beispiel mag hier das beliebte *Request/Reply* Paradigma dienen. In diesem Paradigma holt sich ein Klient über eine definierte Punkt-zu-Punkt Verbindung von einem Dienstanbieter Daten. In ubiquitären Umgebungen ist eine solche Form der Interaktion nur noch beschränkt verwendbar:

- *Flüchtige Bindungen.* In einem *Request/Reply* Szenario ist es für den mobilen Klienten notwendig, die Adresse und den Port des Dienstes zu kennen von dem er Daten abrufen kann. In den hier betrachteten Szenarien sind genau dies Informationen, die jeweils von Umgebung zu Umgebung wechseln können. Da keine globale administrative Instanz angenommen werden kann, ist hier ein hohes Maß an Heterogenität wahrscheinlich. Abhilfe können dedizierte Protokolle und Verfahren schaffen, die solche Meta-Informationen „entdecken." Als Beispiel sei hier Jini [Sun99a] von Sun Microsystems genannt. Der nachgewiesene Nachteil dieser Verfahren ist der hohe Resourcenverbrauch auf mobilen Geräten, welche den Nutzen im ubiquitären Fall zweifelhaft erscheinen läßt. Daher erscheint ein leichtgewichtiger und auf die eigentlich benötigten Daten ausgerichteter Ansatz wesentlich besser geeignet für die genannten Anwendungsfälle. Dieser Ansatz wird auch im Rahmen dieser Dissertation verfolgt.

- *Enge Kopplung.* Als weiteres Phänomen der hohen Mobilität und Spontanität der Interaktion ist, dass auf der Verbindungsebene Annahmen über das „normale" Verhalten der Kommunikationspartner in einen *Request/Reply* Szenario fehlschlagen. Dort wird normalerweise angenommen, dass der Verlust der Verbindung zwischen Klient und Dienst automatisch mit einem Fehlerzustand gleichzusetzen ist. Eine solche *enge Kopplung* ist jedoch in einem mobilen System i.d.R. nicht realisierbar. Mobile Klienten, die ihre Energie aus einem Akku beziehen, werden versuchen möglichst Energie zu sparen und sich regulär ausschalten. Oder sie geraten in ein „Funkloch" und sind für eine Weile nicht erreichbar. In einem eng gekoppelten System wird dies unweigerlich zu einem Fehler führen. Daher ist es notwendig, die grundlegenden Eigenschaften der Verbindungsebene zu berücksichtigen und integral mit in das Model der Kommunikation zu integrieren. In der vorliegenden Arbeit wird dies im Rahmen des Entwurfs eines verteilten Notifikationsdienstes unter dem Aspekt *lose gekoppelte Systeme* berücksichtigt.

Ein weiterer zu berücksichtigender Aspekt ist, dass ein Ressourcen-limitiertes Gerät sehr leicht vom eigentlichen Management seiner eigenen Mobilität überfordert werden kann. Allein die Integration in eine neue Umgebung kann schon zu komplex sein, wie z.B. Jini negativ beweist. Weiterhin ist zu beachten, dass in hochgradig vernetzten System mit einer großen Anzahl von Artefakten auch die Menge der Daten, die sich zu einem gegebenen Zeitpunkt im System befinden, stark ansteigt. Ein „kleines" Gerät kann folglich unbrauchbar werden, wenn es selbst für die notwendige Filterung sorgen muss. Darüber hinaus muss ein mobiles Gerät möglicherweise einige oder sogar alle Aspekte

von Mobilität in die Infrastruktur delegieren und so „mobilitätstransparent" agieren. In dieser Situation ist zu überlegen, ob eine „Mobilitätsschicht" zur Vermittlung zwischen mobilem Gerät und der eigentlichen Infrastruktur vorteilhaft ist.

Adaptives Verhalten. Eine wichtige Klasse von Anwendungen in ubiquitären Szenarien ist diejenige der kontextbewussten und kontextadaptiven Anwendungen. Im Allgemeinen zeichnen sich diese Anwendungen dadurch aus, dass sie sich der jeweiligen Umgebung anpassen und auf Änderungen ihrer Ausführungsumgebung *reagieren*. Hier ist anzumerken, dass solche Änderungen typischerweise extern von der Applikation stattfinden, d.h., in der (physischen) Umgebung in der sich ein mobiles Gerät momentan befindet. Ein einfaches Beispiel ist das Ausschalten akustischer Signale, wenn sich ein Benutzer in der realen Welt in einem Meeting befindet. Die Applikation *adaptiert* ihr Verhalten dann entsprechend der aktuell detektierten Situation. Die Herausforderungen auf diesem Gebiet können folgendermaßen charakterisiert werden: (i) Wie kann *Kontext* auf syntaktischer *und* semantischer Ebene von Applikationen aus der umgebenden Infrastruktur bezogen werden, insbesondere wenn man typische Probleme von Heterogenität und Anzahl der (Roh-)Datenquellen in der Umgebung in Betracht zieht. (ii) durch das Fehlen einer zentralen Administration ist die Frage von Bedeutung, wie verteilte unabhängige Komponenten zu einem bestimmten Teilsystem orchestriert werden können, so dass sie zur Erreichung eines bestimmten Applikationsziels beitragen. Das Hauptproblem ist in den unterschiedlichen Domänen von Infrastruktur und Applikationen zu suchen. Während eine Infrastruktur per Definition möglichst unabhängig von den einzelnen Klienten operieren soll und daher hauptsächlich abhängig von den Daten operiert, die sich aktuell im System befinden, ist eine kontextabhängige Applikation sehr domänenspezifisch. Sie besitzt ein konkretes Applikationsziel und ist abhängig von spezialisierten Daten, welche ggf. erst aus Rohdaten des Systems extrahiert werden müssen.

Entkopplung in Raum und Zeit. Wir haben bereits darauf hingewiesen, dass *lose gekoppelte Systeme* in vielen Bereichen der engen Koppelung im vorliegenden Anwendungsfeld überlegen sein können und deshalb den Fokus dieser Arbeit bilden. Im Bereich der lose gekoppelten Systeme gibt es bereits einige Arbeiten, die dieses Paradigma in zumeist klassischen verteilten Systemen unterstützen. Hier sind exemplarisch zu nennen das ungerichtete Senden („*broadcasting*") von Daten in einem System [WC02; AFZ97], geteilte Daten- oder Tupelräume [RC90; ACG86; CG89] und auf Ereignissen basierende Publish/Subscribe Systeme [SA97; CRW01; CDF01; Müh02; FMG03]. Alle diese Ansätze haben gemeinsam, dass die Produzenten von Daten kein a priori Wissen über die Konsumenten, d.h. Empfänger, dieser Daten haben müssen (und umgekehrt). Beide Parteien sind unabhängig voneinander und in der Regel gegenseitig anonym. Alle oben genannten Implementierungen für lose gekoppelte Systeme sind Beispiele für zwischengeschaltete Infrastrukturen, die dafür Sorge tragen, dass Sender und Empfänger ohne explizites Wissen über den Anderen miteinander interagieren können. Die jeweilige Infrastruktur ist dafür verantwortlich, Daten von der Quelle zu den jeweiligen Senken zu transportieren. Die Interaktion zwischen Sendern und Empfängern ist deshalb auch typischerweise *datengesteuert*. Ein solches Verhalten ist inbesondere interessant für die in der vorliegenden Arbeit betrachteten Szenarien, in denen das „Was" wichtiger für Applikationen ist als das „Wer" das Ereignis beobachtet hat. Interaktion ist also inhärent datengesteuert.

Als weiterer Faktor kommt hinzu, dass die Zahl der Konsumenten von Daten als wesentlich größer angenommen werden darf als die Zahl der Produzenten. In den großskalierenden Systemen, die wir hier als Beispiel nehmen, ist darum die Notwendigkeit zur effizienten *eins-zu-viele* oder *viele-zu-viele* Kommunikation gegeben („Multiplexer"). Als Beispiel sei hier ein einfacher Temperatursensor

genannt, dessen Sensordaten für mehr als eine Applikation interessant sein können. Dies alles begründet die Notwendigkeit der Entkopplung „im Raum."

Zusätzlich ist eine Entkopplung in der Zeit notwendig. Dies lässt sich leicht aus den bereits genannten Charakteristika mobiler Systeme ableiten. Oftmals ist es für einen potenziellen Empfänger einer Nachricht schlicht nicht möglich zum normalen Empfangszeitpunkt auch wirklich empfangsbereit zu sein. Sei es weil das Gerät zu diesem Zeitpunkt ausgeschaltet ist oder weil es anderweitig nicht erreichbar ist. Auch hier kann eine entsprechend ausgestattete Infrastruktur wesentliche Funktionen dieser Entkopplung zur Verfügung stellen.

Effiziente Verteilung von Nachrichten in mobilen Systemen. Ein weiteres Merkmal der in dieser Arbeit betrachteten ubiquitären Systeme ist die Notwendigkeit einer effizienten und skalierbaren Nachrichtenverteilung. Solche Systeme können zu sehr großen und komplexen Systemen skalieren. Dies kann anhand zweier Dimensionen geschehen: (i) der reinen physikalischen Größe von global verteilten vernetzten Systemen, die möglicherweise eine ganze Stadt oder sogar den gesamten Globus umspannen und (ii) der inhärenten Komplexität. (ii) bezieht sich auf die immer höhere Integration von elektronischen Komponenten und der damit einhergehenden steigenden Anzahl von Prozessoren in immer mehr alltagsweltlichen Artefakten. Auch in (physikalisch) kleinen Umgebungen können sich so eine große Anzahl vernetzter System finden lassen. Als Beispiel kann hier ein PKW dienen, in dem mehrere hundert Kleinst-Computer ihren Dienst versehen. Ein großskalierendes System muss daher nicht notwendigerweise auch ein großes (physikalisches) Gebiet beanspruchen. Als Konsequenz aus den vorhergehenden Betrachtungen kann postuliert werden, dass eine Infrastruktur mit einer sehr großen Anzahl von Daten umgehen können muss. Viele Produzenten produzieren Daten für viele Konsumenten. Solche Daten „schweben" dann in großer Anzahl durch ein vernetztes System. Daraus resultiert die Notwendigkeit der effizienten Behandlung dieser Daten, sowohl auf der Ebene der Verteilung als auch der Ebene des Managements. Hier sei als Beispiel das Problem des „Eigentums" von Daten genannt und insbesondere die verteilte *Garbage Collection*, also das kontrollierte Entfernen von Daten aus dem Gesamtsystem.

Als die obige Diskussion überspannende Beobachtung kann festgestellt werden, dass ubiquitäre Systeme sich schlecht auf eine spezielle Problem-Domäne eingrenzen lassen. Das Gegenteil ist der Fall. Wir müssen einen stark holistisch geprägten Ansatz für die Unterstützung solcher Systeme wählen. In der vorliegenden Arbeit stellen wir uns dieser Herausforderung und geben eine Architektur für einen veteilten Notifikations-Dienst für ubiquitäre Umgebungen an, der die begenannten Anforderungen adressiert und in diesem Kontext neuartige Lösungen entwickelt.

Ergebnisse dieser Arbeit

Der Ansatzpunkt dieser Dissertation ist die Vision der zukünftigen und hochgradig vernetzten ubiquitären Systeme. Benutzer dieser Systeme werden in ihrer Dienstenutzung andere Schwerpunkte haben als das in heutigen Systemen beobachtbar ist. Eingebettet in Umgebungen, die eine Vielzahl nützlicher Dienste anbieten, werden Dienstnutzungszenarien mit entsprechenden Zugriffsmechanismen geprägt sein von einem hohen Maß an Spontanität. Benutzer greifen in solchen Szenarien spontan und ad-hoc auf sie umgebende Dienste zu, abhängig von ihrer jeweiligen Situation. Hier spielen lokations-basierende Dienste, also Dienste die angepasst an die jeweilige Position ihre Dienste erbringen, eine herausragende Rolle. Wir glauben, dass Benutzer sich dennoch dabei auf personalisierte mobile Geräte verlassen werden, die vom jeweiligen Benutzer in die jeweilige Umgebung eingebracht werden (vgl. „Pervasive Computing," weiter oben).

Daher ist es notwendig, unterschiedlichste mobile Geräte effizient zu integrieren und zu unterstützen. Ein Beispiel dafür ist die effiziente Lieferung von Informationen an den jeweiligen Benutzer bzw. dessen Gerät. Andere essentielle Herausforderungen dieses Modells haben wir bereits im vorangegangenen Abschnitt charakterisiert.

In dieser Arbeit argumentieren wir, dass im Hinblick auf diese Herausforderungen ein Teil der Lösungen zu den genannten Problemen in die Infrastruktur angesiedelt sein müssen. Daher ist es notwendig, die Infrastruktur zu einem gewissen Grade an diese neuen Herausforderungen anzupassen, was zu neuartigen Anforderungen und Lösungen für eine Infrastrukturunterstützung führt.

Dies erfordert ein grundlegend verändertes Systemmodell, verglichen mit „klassischen" verteilten Systemen. Dort kann das System aus verlässlichen und konkreten Annahmen über das Verhalten der jeweiligen Komponenten aufgebaut werden. Übliche Annahmen sind: Stationäre Klienten, verlässliche, breitbandige und dauerhafte Netzwerkverbindungen untereinander, sowie eine gewisse Konstanz der Interaktions-Beziehungen, d.h. der Kontext der Verarbeitung verändert sich wenig bis gar nicht. Darüber hinaus wird oftmals auch die Existenz einer zentralisierten Administration angenommen. Im Lichte dieser Annahmen folgen daher bestimmte Designentscheidungen, die wir hier als *Black-Box Modell* bezeichnen wollen. Effektiv wird die Existenz eines verteilten Systems in der Infrastruktur-Schicht „versteckt," so dass sich viele Aspekte des Systems genauso wie im nicht-verteilten Fall verhalten. Mehr noch, komplette Teile einer Applikation können z.T. dauerhaft in die Infrastruktur ausgelagert werden. Dieser Ansatz ist basierend auf den Annahmen valide.

Wir argumentieren hingegen, dass ein solcher Ansatz im betrachteten Anwendungsfall nicht länger gültig ist und daher scheitert. In ubiquitären Systemen findet Evolution nicht graduell statt, sondern rapide. Komponenten verändern sich oft und sind dezentral gesteuert. Klienten und auch Dienste sind mobil und daher als kurzlebig anzusehen. Kontrolle findet nicht oder nur dezentral statt. Große Teile des Systems wie insbesondere die Netzwerkverbindungen sind als unzuverlässig einzustufen. Dementsprechend muss Interaktion i.W. opportunistisch und lose gekoppelt sein.

Um diesen Paradigmenwechsel zu dokumentieren, ist ein wesentliches Ergebnis von Kapitel 2 die eingehende Analyse der Defizite einer klassischen Middleware im Kontext der ubiquitären mobilen Systeme. Teil der Analyse ist die Erstellung einer detailierten Anforderungsliste für die Erweiterung klassischer Infrastrukturen im Hinblick auf den Einsatz in ubiquitären Szenarien. Damit einhergehend zeigt Kapitel 3, dass eine aussichtsreiche Basis für eine solche evolutionäre Erweiterung das bekannte *Publish/Subscribe* Paradigma ist. Wir zeigen klar auf, auch im direkten Vergleich mit anderen Infrastruktur-Paradigmen, welche Anforderungen erfüllt sind und wo sich Lücken zeigen, die diese Arbeit zu füllen sucht. Nachfolgende Ergebnisse dieser Arbeit beziehen sich konsequenterweise auf die Ergebnisse aus Kapitel 2 and 3.

Eine Kernaussage dieser Arbeit ist, dass eine angemessene Form der Mobilitätsunterstützung Teil der Infrastrukturaufgaben sein muss. Die heute oftmals vorgefundene Delegation an die Anwendungsschicht macht im Anbetracht folgender Anwendungsszenarien nur eingeschränkt Sinn: (1) „Legacy" Anwendungen, also Anwendungen, die bereits im Einsatz sind, sollen nach wie vor lauffähig sein, sowohl im statischen wie im mobilen Fall; (2) Anwendungen, die explizit die zusätzlich vorhandenen Informationen über Lokalität und Mobilität nutzen wollen oder müssen. Aus (1) folgt direkt, dass die bestehende Schnittstelle zu der Infrastruktur und auch deren Verhalten im mobilen und nicht-mobilen Einsatz gleich erfahrbar sein muss. Aus (1) und (2) zusammen folgern wir, dass der Ansatz der vorliegenden Arbeit, also die evolutionäre Erweiterung eines bestehenden und erfolgreich im Einsatz befindlichen Notifikationsdienstes, der kompletten Neuerstellung einer Infrastruktur vorzuziehen ist. Aufgabe der Infrastruktur ist dann natürlich eine Behandlung der neu hinzugekommenen Anforderungen der Mobilitätsunterstützung. Als Konsequenz können dann An-

wendungen, die in die obige Kategorie (1) fallen, von einem statischen Kontext in einen mobilen Kontext transferiert werden. Hingegen erfordert (2) eine weitgehend automatisierte Behandlung von Lokationswechseln in der Infrastruktur. Dann ist Lokationsinformation unabhängig von einer bestimmten Applikation und nur noch gebunden an die aktuelle Bewegung eines Objektes im realen Raum.

Mobilitätsunterstützung im Kontext eines verteilten Notifikationsdienstes ist daher das zentrale Thema der Kapitel 5, 6 und 7. Flankiert werden die genannten Kapitel durch Grundlagen aus Kapitel 4.

Das Hauptresultat von Kapitel 5 ist eine Lösung für die transparente Einbindung mobiler Klienten in einen verteilten Notifikationsdienst. Das Ausblenden verschiedener Aspekte von Mobilität ist eine übliche Anforderung der Anwendungsebene mobiler Klienten an die Infrastruktur. Im hier betrachteten speziellen Anwendungsfall, zusammen mit einem Notifikationsdienst, ergeben sich darüber hinaus besondere Anforderungen, wenn Klienten Nachrichtenbroker im Netzwerk des Notifikationsdienstes verlassen oder wechseln (engl.: *Roaming*). Ein solcher Wechsel soll möglichst für einen Klienten unspürbar sein. Mobilitäts-Transparenz ist ein erster notwendiger Schritt das erfolgreiche Paradigma des *Publish/Subscribe* vom statischen Anwendungsfall in hoch mobile und dynamische Szenarien abbilden zu können. Wir präsentieren ein neues Verfahren zur dynamischen Weitervermittlung von mobile Klienten an neue Broker, das die genannte Tranzparenz-Anforderungen erfüllt. Das Verfahren erlaubt es, bestehende Anwendungen aus dem bisher angenommenen statischen Umfeld in ein mobiles Umfeld zu transferieren, ohne dabei auf zugesicherte Eigenschaften der unterliegenden Infrastruktur im mobilen Fall verzichten zu müssen. Die Spezifikation unseres Verfahrens beruht auf einer detailierten Analyse der Anforderungen aus Sicht der Anwendungsebene und berücksichtigt dabei andererseits auch die Auswirkungen auf den realisierenden Notifikationsdienst. Das eingeführte Verfahren vereint nahtlos die bestehende Funktionalität des *Content-based Routing*, wie es in der verwendeten Referenzimplementierung unseres Notifikationsdienstes zum Einsatz kommt, mit der Unterstützung einer unterbrechungsfreien und Sender-FIFO geordneten Auslieferung von Nachrichten an mobile Klienten. Damit unsere funktionalen Erweiterungen auch anwendbar sind für Klienten des Notifikationsdienstes, die sich ihrer Mobilitätseigenschaften gar nicht bewusst sind, werden sie vollständig in der Infrastruktur verborgen und bedürfen keiner Änderung der Schnittstelle zwischen Klient und Notifikationsdienst. Das ist besonders wichtig in Szenarien, in denen bereits bestehende Anwendungen in mobilen Umgebungen zum Einsatz kommen sollen. Eine weitere herausragende Eigenschaft unseres Verfahrens ist, dass es sich ausschließlich auf bereits bestehende Funktionalitäten des verwendeten Notifikationsdienstes abstützt. So bedarf es keiner zentralen Datenspeicherung zur nachträglichen Versendung an Klienten, keiner zentralisierten Kontrolle und auch keiner Kommunikation außerhalb der verwendeten *Publish/Subscribe* Infrastruktur. Das spezifizierte Verfahren bildet die zentralen qualitativen Eigenschaften des *Publish/Subscribe* Paradigmas, wie z.B. lose Koppelung, Sender-FIFO oder Vollständigkeit, in angemessener Weise vom statischen Anwendungsfall auf dynamische mobile Anwendungsszenarien ab.

In Kapitel 6 wird die Integration mobiler Klienten in unsere verteilte *Publish/Subscribe* Infrastruktur einen wesentlichen Schritt weitergetrieben. Dort führen wir das Konzept der *lokationsabhängigen* Subskriptionen und des *lokationsabhängigen* Matchings von Notifikationen und Subskriptionen ein. Hier fließt auch das Referenz-Lokationsmodell ein, das wir in Kapitel 4 angeben. Mit Hilfe des Lokationsmodells wird die Infrastruktur in die Lage versetzt, Lokationsbewußtsein von kontextsensitiven Anwendungen in effizienter Weise zu unterstützen. Dabei sind zwei große Herausforderungen zu beachten: Erstens ist die jeweilige Anpassung einer lokationsabhängigen Subskription an den jeweiligen Aufenthaltsort eines Klienten von der Infrastruktur durchzuführen und nicht vom je-

weiligen Klienten. Zweitens ist besonderes Augenmerk auf die Realisierung eines effizienten und unterbrechungsfreien Nachrichtenflusses zu legen. Dabei besteht die Gefahr darin, dass die Nachrichtenzustellung ohne entsprechende Techniken in der Infrastruktur schnell zum simplen Fluten im Netzwerk degeneriert. Das ist jedoch aufgrund der asymetrischen Resourcenverteilung zwischen mobilen Geräten und Infrastruktur auf jeden Fall zu vermeiden. Wir geben daher in Kapitel 6 ein adaptives Verfahren an, mittels dessen Klienten unserer Infrastruktur in die Lage versetzt werden, ihr Interesse an lokationsspezifischen Informationen auf eine definierte Weise zu formulieren und an den Notifikationsdienst weiter zu geben. Lokationsabhängige Subskriptionen, die so in das System gelangen, werden danach von diesem gepflegt, angepasst und entsprechend der Lokationsspezifikation ausgewertet. So gelangen nur aktuelle und zutreffende Informationen zum Klienten, ohne das jener seine Subskriptionen jedesmal „manuell" anpassen muss. Ebenfalls automatisch angepasst wird ein spezieller Platzhalter, der in Subskiptionen verwendet werden kann: `MyLoc`. Er bezieht sich jeweils auf die aktuell feststellbare Position eines Klienten und dient dazu, die *räumliche* Gültigkeit einer Subskription in der realen Welt zu definieren. Dennoch ist es unabwendbar, dass ein Klient erst mit Verzögerung oder gar nicht lokalisiert werden kann. Daher ist es sinnvoll und notwendig, eine gewisse *Unschärfe* der Klienten-Positionierung bei der Nachrichtenzustellung zu verwenden. Das ergibt sich ebenso aus dem Wunsch nach akkurater und zeitnaher, d.h. im Idealfall unterbrechungsfreier Zustellung von lokationsabhängiger Information. Die einzige uns bekannte Alternative ist das erwähnte Fluten von Nachrichten im Netzwerk. Um Fluten zu vermeiden, verwenden wir adaptive Filter, die den Bereich der Nachrichtenzustellung anhand eines Bewegungsgraphen einschränken und so effektiv ein Fluten vermeiden.

Dennoch reichen beide bisher eingeführten Lösungen nicht aus, ein spezifisches Problem der asynchronen Kommunikation in mobilen Systemen zu lösen. Aufgrund der Entkopplung zwischen Sender und Empfänger von Nachrichten kann der Klient nicht explizit notwendige Informationen zur eigenen Initialisierung von einem Produzenten anfordern. Normalerweise wird hier angenommen, dass ein System eine gewisse Latenzzeit aufweist, bis es sich eingeschwungen hat. In manchen Umgebungen kann diese Latenzzeit größer sein als die Zeit, die ein mobiler Klient dort verbringt. Außerdem widerspricht ein solches Verhalten dem Wunsch eines Benutzers nach spontaner Interaktion mit dem System. Gesucht ist also ein weiteres Verfahren, dass es ermöglicht, die Einschwingphase zu minimieren, auch wenn das Gesamtsystem lose gekoppelt ist. In Kapitel 7 geben wir daher ein Verfahren an, das die Eigenschaften der beiden vorangegangenen Verfahren in einer Weise kombiniert, dass ein Klient seine Latenzzeit minimieren kann. Im Allgemeinen braucht eine Anwendung eine gewisse Anzahl an Notifikationen, um einen konsistenten Zustand zu erreichen, von dem ausgehend eine normale Operation möglich ist. Der Zeitraum, über den sich diese passive Phase des „Zuhörens" erstreckt, ist aufgrund der asynchronen Natur des Gesamtsystems nicht vorhersehbar. Wir adressieren das Problem mit einem infrastrukturbasierten Ansatz und plazieren Pufferspeicher innerhalb des Broker-Netzwerks derart, dass Klienten auf Notifikationen zugreifen können, die bereits in der Vergangenheit durch das Netzwerk propagiert wurden und im normalen Modell des *Publish/Subscribe* bereits aus dem System gelöscht sind. Um die Systemlast zu minimieren, geben wir darüber hinaus noch eine Anzahl von Such- und Zusammenführungsstrategien für den Zugriff auf diese Pufferspeicher an. Zusammen genommen kann ein Klient auf diese Art und Weise vollkommen transparent auf Informationen zur Initialisierung zugreifen und so i.d.R. seine Latenzzeit zur Aufnahme seiner normalen Operationen minimieren. So konnten wir einen wichtigen Beitrag zur weitgehenden Entkopplung von Sendern und Empfängern in Raum *und* Zeit liefern.

Ein wesentliches Resultat von Kapitel 2 ist die Identifikation und Klassifikation von Kontext als wichtiger Quelle für angepasstes kontextsensitives Verhalten von Applikationen. Die Anpassungen

sind typischerweise reaktiv, ausgelöst durch Veränderungen des externen Ausführungskontexts. In Kapitel 8 zeigen wir, wie sich ein solches Verhalten auf allgemeine Entwurfsmuster für kontextsensitive Anwendungen auswirkt und geben einen Entwurfsprozess an, der diesem Rechnung trägt. Als Grundlage verwenden wir dazu das Paradigma der *kontrollbasierten Koordination*. Als Konsequenz wird die abstrakte Applikationssemantik in Form von Finiten Automaten gekapselt und getrennt von der Aquisition der Eingabedaten für die Automaten behandelt. Im Wesentlichen lehnt sich der von uns vorgeschlagene Entwurfsprozess an Mechanismen an, wie sie auch im Paradigma des *Model Driven Development* vorzufinden sind. Unser Entwurfskonzept zeigt, wie die Spezifikation einer Applikation in einem definierten Prozess auf entsprechende Subskriptionen abgebildet werden kann, die wiederum zur Laufzeit auf den eigentlichen Daten in einem System operieren und so die notwendigen Eingaben für die Operation einer Anwendung liefern. Dies beinhaltet insbesondere auch die Verwendung wohldefinierter *Eventoperatoren*, wie Aggregation und Komposition von Notifikationen.

Fazit

Die vorliegende Arbeit entwickelt auf der Basis eigener Einschätzungen mehrere wichtige und neue Erweiterungen eines etablierten Paradigmas verteilter Systeme, dem *Publish/Subscribe* Paradigma, zu einer effizienten und einfach anzuwendenden Infrastruktur-Basis für die Verwendung in mobilen ubiquitären Umgebungen. Wir haben die Schlüsselfaktoren analysiert, anhand derer sich ubiquitäre Szenarien fundamental von konventionellen verteilten Systemen unterscheiden. Darauf aufbauend haben wir gezeigt, dass *Publish/Subscribe* ein vielversprechender Ausgangspunkt für eine Infrastruktur ist, die bereits einige der wesentlichen Aspekte ubiquitärer Systeme adressiert. Auf der anderen Seite konnten wir zeigen, dass konventionelle Notifikationsdienste nicht ausreichend sind, um alle genannten Schlüsselaspekte abzudecken. Wir sind überzeugt, dass unser daraus abgeleiteter Ansatz der evolutionären Erweiterung bestehender Funktionalitäten einer Infrastruktur normalerweise einer kompletten Neuerschaffung vorzuziehen ist. Daher haben wir besonderes Augenmerk darauf gerichtet, Verfahren angegeben, die *explizit* Gebrauch von Funktionalitäten machen, wie sie in heutigen Systemen bereits vorzufinden sind. Als Basis dazu diente ein bereits existierender verteilter Notifikationsdienst, der ein *Content-based Routing* Verfahren verwendet. Von diesen Grundfunktionalitäten ausgehend haben wir allgemeine Verfahren angegeben, die insbesondere wichtige Probleme der Unterstützung mobiler und kontextsensitiver Klienten lösen. Trotzdem sind alle Verfahren allgemein genug, um auch auf andere Notifikationsdienste anwendbar zu sein. Ebenso haben wir in dieser Arbeit die Auswirkungen auf den allgemeinen Programmentwurf für kontextabhängige Anwendungen untersucht, um so den gesammten Bogen von der Behandlung von Rohdaten bis hin zur Generierung von Kontext abzudecken. In unserer Überzeugung stellt eine verteilte *Publish/Subscribe* Infrastruktur, zusammen mit den von uns angegebenen neuen Verfahren eine neue, wichtige und dem Anwendungsgebiet angepasste Lösung für die Instrastrukturunterstützung in mobilen und ubiquitären Umgebungen dar.

Danksagung

New ideas pass through three periods:
It can't be done.
It probably can be done, but it's not worth doing.
I knew it was a good idea all along!

Arthur C. Clarke, Science-Fiction writer

Das Verfassen einer Dissertation erfordert – ohne Zweifel – ein hohes Maß an Motivationsfähigkeit, Durchhaltevermögen und Zeit. Nur zu oft sitzt man am Schreibtisch und fragt sich: „Warum tue ich mir das eigentlich an?" Diese Zeiten sind es, in denen man all die Menschen braucht, die dazu beitragen, dass die Arbeit über die Zeit Formen annimmt und Inhalt bekommt. Andererseits ist es unglaublich schön, wenn schließlich Form und Inhalt mehr und mehr erkennbar werden (auch wenn jede Frage, die beantwortet wurde, zehn neue aufwirft).

Naturgemäß begleiten so manche Irrungen und Wirrungen den Weg einer Dissertation. Dann bedarf es dringend „externer Meinungen" und Anleitung diese aufzuklären und zu entwirren. Dazu kommt die Frage nach einem interessanten und spannenden Themengebiet. Für das eine wie das andere danke ich den beiden Professoren, die den größten Anteil am Erfolg meiner Promotion haben: Prof. Alejandro Buchmann und Prof. Friedemann Mattern.

Darüber hinaus sind da natürlich auch all die anderen Menschen, ohne die es einfach nicht „gegangen" wäre. Aber wo anfangen?

Derjenige, dem ich zu extrem großem Dank verpflichtet bin, ist Ludger Fiege. All die Diskussionen, die wir in all den Monaten geführt haben, waren absolut essentiell für mich. Nebenbei bemerkt haben sie auch viel Spaß gemacht! Seine Geduld und Offenheit, aber auch seine Standhaftigkeit in Diskussionen, sind nur einige seiner bestechenden Merkmale. Danke für all die Zeit und die Geduld.

Dann ist da natürlich noch Roger Kilian-Kehr. Mit ihm habe ich immer viel und gerne zusammengearbeitet. Unsere Diskussionen haben mir schmerzlich gefehlt (und fehlen mir noch), nachdem er promovierte und den Lehrstuhl verlies. Auch Dir meinen Dank.

Allen anderen Kollegen auf dem Weg würde ich sicher und gerne eine ausführliche Widmung ihrer jeweiligen Verdienste schenken, aber das wäre dann doch zu lang. Deshalb, ich danke euch allen! Jedem einzelnen und allen gemeinsam. Zuerst sind da alle meine Kollegen des Fachgebiets „Datenbanken und Verteilte Systeme" zu nennen. Dann auch Dank an die Kollegen des Graduiertenkollegs „Enabling Technologies for Electronic Commerce" und die des Information Technology Transfer Office's (ITO). Ebenso Dank an Felix Gärtner und Oliver Kasten für die tolle Zusammenarbeit.

Ein besonderer Dank gilt unserer "guten Seele" Marion Braun, ohne die das Leben an unserem Fachgebiet nicht denkbar wäre.

Ebenfalls zu Dank verpflichtet bin ich den Korrekturlesern dieser Dissertation Felix Gärtner, Roger Kilian-Kehr und Behnam Rassekh für ihre wertvollen Hinweise und Kommentare.

Ein ganz, ganz besonderer Dank gilt meiner Mutter, denn ohne sie würde ich diese Zeilen an dieser Stelle sicher nicht schreiben. Sie, wie auch meine Geschwister, haben stets die Sicherheit gehabt, dass ich das „schon schaffen werde". Danke für das Vertrauen!

Da ich einiges von der Arbeit mit in mein Privatleben genommen habe, danke ich all meinen Freunden dafür, dass sie für die notwendige Ablenkung gesorgt haben, mich „geerdet" haben, wann immer notwendig, und mich unterstützt und aufgemuntert haben, wenn ich mal alles hinschmeissen wollte. Danke!

Eine Person ist noch besonders hervorzuheben. Sie hat einen unermesslich großen Anteil am Erfolg dieser Arbeit. Diese Person ist Cornelia Brödel. Sie hat mich auf meinem Weg lange Zeit begleitet und mich dabei über alle Maßen unterstützt. Dafür gehört ihr mein ewiger Dank.

Jetzt, da ich langsam anfange im Rückblick auf diese Zeit der Promotion zu schauen, kann ich nur festhalten, dass der Weg vom Anfang bis zur Beendigung mit Sicherheit nicht eine Gerade ist. Aber auch jede Wirrung hat ihren Nutzen. Das weiß man aber meist erst hinterher. Es hat viel Spaß gemacht, aber wie oft war ich mir sicher, dass ich das alles „canceln" möchte. Immer wieder die notwendige Motivation zu finden ist deshalb wohl der wahre Schlüssel und die wahre Lehre, welche zu ziehen ist.

Dennoch, zu jedem Erfolg tragen immer auch (und insbesondere) viele Menschen bei. Deshalb nochmals mein Dank an alle meine Kollegen, meine Familie und meine Freunde. Danke!!!

Contents

List of Figures

List of Tables

List of Acronyms

CEP	Complex Event Processing
CEA	Cambridge Event Architecture
CORBA	Common Object Request Broker Architecture
DAG	Directed Acyclic Graph
DNS	Domain Name Service
DTD	Document Type Definition
ERP	Enterprise Resource Planning
FIFO	First-in, First-out
FSM	Finite State Machine
GPS	Global Positioning System
GSM	Global System for Mobile Communication
LBS	Location-based Service
IETF	Internet Engineering Task Force
IR	Infrared
JMS	Java Message Service
JNDI	Java Naming and Directory Interface
JVM	Java Virtual Machine
LAN	Local Area Network
MBONE	Internet Multicast Backbone
MIME	Multipurpose Internet Mail Extensions
P2P	Peer-2-Peer
PC	Personal Computer
PDA	Personal Digital Assistant
RF(ID)	Radio Frequency (Identification)
RFC	(Internet) Request for Comments
RMI	(Java) Remote Method Invocation
RPC	Remote Procedure Call
TTL	Time-to-Live
URL	Uniform Resource Locator
UML	Unified Modeling Language
WAN	Wide Area Network
WAP	Wireless Application Protocol
WGS84	World Geodetic System
WLAN	Wireless Local Area Network
WSN	Wireless Sensor Networks
XML	Extensible Markup Language

1 Introduction

As for the future,
your task is not to forsee it, but to enable it.

From "Citadelle" by Antoine de Saint-Exupery, Poet and Pilot (1900-1944)

1.1 Motivation

Throughout the last years we have observed the tremendous success of mobile telephony and, more recently, portable devices, like *personal digital assistants* (PDA) and laptop computers. Bundled together with wireless communication interfaces and environmental sensors (e.g., GPS), many expect them to be a major technological trend and driving force for the next decade, both for research and economy. Many researchers and analysts even believe that not only get devices mobile and reasonably powerful but that the surroundings of these devices themselves get filled with all kinds of communicating devices. This vision is called *ubiquitous computing* [Wei91; Wei93] or *pervasive computing*. Especially pervasive computing, as used by IBM [IBM01b], focusses on technologies necessary to enable users to seamlessly interact with the surrounding physical and social environment. The central underlying assumption is that users will always prefer a *personalized* (and hence trustworthy) mobile device as the enabling technology for such interaction. Part of the interaction can be the access to local resources found in the current environment, such as a printer-service, but more important is the adaptation of applications to the current physical and executional context the device is located in.

"Adaptation" is an important aspect of what usually is called "context-aware" or "situation-aware" computing. Users of mobile devices interact with the current environment and at the same time expect their devices to become "smart", i.e., in changing situations a changing behavior is expected. Part of this "smartness" of applications is *awareness* of the environment, demanding information about the current computational as well as physical context. Information which is not part of a mobile application or the mobile device, but is only available from the current (and ever changing) infrastructure the device is located in. Probably the most prominent example for such context-aware behavior are *location-based information services*, where the content provided by the service is dependent on the actual position a device is located at.

However, on the advent of pervasive computing, where mobile devices will be interacting with surroundings filled with all kinds of sensors, actuators, and other smart devices, we see a number of fundamental key challenges:

Mobile systems. Obviously, mobility of users and their digital appliances is a major driving force for pervasive systems. Clients and services move around freely. This introduces a significant need to handle the inherent dynamics appropriately. Due to client mobility certain assumptions about

possible modes of interaction fall short. For instance, the classical approach of using *request/reply* or *point-to-point connections between ports* is not well suited, for the following reasons:

- *Volatile bindings.* For a mobile device it is necessary to know the name and address of a resource it wants access to. Unfortunately, in changing environments (local) resources usually are not bound to the same addresses. Hence, resources must be "discovered" before they can be used. Some resource discovery schemes were proposed, e.g., Jini [Sun99a], but often resource consumption on the device is considerably high and a lightweight and data-driven approach is more desirable. Association and coordination of entities must be built-up on-the-fly.

- *Tight coupling.* Phenomena like connection loss or unexpected behavior of one communication partner is considered an erroneous state in the request/reply paradigm and is not assumed to happen often. Unfortunately, this exact behavior is common in mobile systems and must be integrated into the model of interaction. Clients might move out of communication range of a wireless connection or are simply powered off. To deal with the unpredictability of mobile environments, a different paradigm than request/reply seems to be more adequate.

Moreover, handling the implicit complexity of joining and leaving a local environment easily can overwhelm a small, resource-limited device. In such situations an additional layer of indirection can be considered beneficial for the versatility of the overall system. This can cater for a small device's demand for *transparency* of mobility. Certain aspects of mobility then are delegated into the infrastructure. On the other hand, *awareness* of mobility can be a valuable input for context-aware applications. Then, the infrastructure should provide means such that applications can draw from this awareness.

Adaptive behavior. An important class of applications for pervasive computing systems are *context-sensitive* or *context-aware* applications. In general, these applications change their behavior dependent on the current execution environment. Usually, such change in behavior is reactive in nature. Often, it is triggered by events occuring in the physical environment, like a person entering a room. The application then *adapts* its behavior in order to cope with the new situation or context. For example, the user may be notified when a shop is in the vicinity offering an item on a shopping list on the device. The challenging questions here are: (i) how can "context" on the level of the *syntax and semantics* of an application be extracted from the various and heterogeneous raw-data items produced by the innumerable sensors and devices around, and (ii) how can distributed and independent components be orchestrated for a task-oriented application to contribute to the application's goal. The complicating issue here lies in the different domains of infrastructures and context-dependent applications: while the former is supposed to stay general-purpose, and thus is driven only by the data coming into the system, the latter usually is special purpose, has a thread of control and is goal oriented, driven by interpreting base data.

Decoupling in space and time. As mentioned above, asynchronous communication as underlying interaction paradigm is favorable over tightly coupled systems in the domain of pervasive computing systems. A possible choice is *loose coupling* as it can be found in broadcasting data [WC02; AFZ97], shared data- or tuplespaces [RC90; ACG86; CG89], or publish/subscribe systems [SA97; CRW01; CDF01; Müh02; FMG03]. All approaches have in common that producers of data, e.g., sensors, and consumers are decoupled and anonymous to each other. An interposed infrastructure takes care of delivering data from producers to consumers. Additionally, for adaptation

to changing environments, the “What” has happend is more important than “Who” has produced the data. Often, the identity of the producer is not important for the execution of an application. Thus, interaction gets much more data-driven. In the settings we consider in this thesis, it can further be assumed that the number of consumers of data is larger than the number of producers. This introduces the need for efficient one-to-many and many-to-many communication. For example, sensor data of a temperature sensor probably can contribute to more than a single application’s thread of execution. This constitutes the need for decoupling in space.

Moreover, as we cannot assume that sender and receipient of data are online permanently, we have to strive for effective means to achieve a resonable decoupling in time as well. Especially in mobile settings where disconnectedness and offline operation is common, a mobile client needs facilities in the infrastructure to re-establish a consistent state for commencing its operation.

Efficient Information Dissemination in Mobile Systems. Efficiency and scalability of information dissemination is a crucial requirement for large-scale mobile or ubiquitous systems. As mentioned above, many-to-many communication is a first step towards efficient information dissemination in such settings. However, pervasive and ubiquitous computing systems can scale along two dimensions. First, in their physical extend, like whole cities or even the globe; Second, in their computational complexity. An environment filled with communicating, networked digital appliances does not have to grow to a large physical size to be considered large-scale.

Consequently, the number of data items in the system can be assumed to be considerably large. Large amounts of single data items are “floating” through the system. For instance, sensors often publish their data readings periodically, like a temperature sensor publishing data readings every few seconds. This imposes the need for efficient means for data dissemination as well as data management. For example, a distributed garbage collection is hard to conduct in anonymous and decoupled settings.

Together, the key challenges of pervasive computing systems are not focussed on a single problem domain. The opposite is the case. Therefore, a more holistic view on supporting such systems is needed. In this thesis we accept this requirement and provide solutions to the named challenges on the level of an architecture for a distributed notification service for pervasive computing environments.

1.2 Contributions of this Thesis

The starting point of this thesis is the vision of future mobile and pervasive computing scenarios. Users are likely to use and access services and data “spontaneously” and ad hoc depending on their current situation. Location-dependent services, i.e., services which are only available at a particular site, play a central role in such scenarios. Service usage will take place to a significant extent through mobile devices the users carry around.

Hence, integration of and efficient information delivery to mobile devices is of outstanding importance. In the previous Section 1.1 we introduced the fundamental challenges for supporting mobile and pervasive scenarios.

In this thesis we argue that in the face of the challenges stated above many of the problems involved require solutions integrated into the core of a strong supporting middleware. In static distributed systems a middleware can be built around concrete assumptions about its clients. Usually,

this includes *fixed* devices, *permanent* and *reliable* network connections, and a *static* context. Evolution takes place slowly and only gradually. Naturally, this leads to the rationale of a *black box model*. Effectively, the side-effects of distribution are hidden in the infrastructure. Even complete parts of applications easily can be relocated into the middleware for execution. We argue that this approach of a black box is likely to fail in mobile and pervasive settings. Evolution takes place rapidly, devices are mobile, and reliable network connections are hard to maintain. A central contribution of Chapter 2 therefore is the thourough analysis of the shortcomings of traditional middleware in the face of the advent of mobile, pervasive settings. It leads to a detailed list of requirements for middleware extensions in order to facilitate for proper support. Additionally, in Chapter 3 we show that a promising candidate for a basis for extensions is the successfully deployed publish/subscribe paradigm. We clearly state which requirements are met when employing a distributed publish/subscribe notification service, like the REBECA notification service, for interaction and where the publish/subscribe paradigm itself, and therefore REBECA, falls short. Subsequent contributions of this thesis address these shortcomings and are centered around the challenges as introduced in Section 1.1.

We argue that support for mobility should be an issue of the publish/subscribe middleware itself and not be delegated to the application layer. Three kinds of application scenarios have to be supported: i) existing applications in a static environment, ii) existing applications in a mobile environment, iii) mobility-aware applications. Since publish/subscribe systems and applications have been deployed very successfully, extending existing systems and models is preferred to creating new "mobile" middleware from scratch. As a consequence, the middleware must transparently handle some of the new mobility issues. This allows existing event-based applications to directly interact with and even to be deployed as mobile applications. On the other hand, the third scenario requires the middleware to support a (semi-)automated handling of location changes. If no such support is available, mobility is actually controlled by the application and not by the movement of the client.

We provide solutions for the two different and orthogonal types of mobility. The first type of mobility is support for location transparency, where clients may temporarily disconnect from the pub/sub system (due to power-saving requirements or the network characteristics). This means that applications are not necessarily aware of the fact that the client is moving, allowing existing applications to be transferred to mobile environments. This is the main contribution of Chapter 5 together with the foundations laid in Chapter 4.

The second type of support is for mobility-aware applications, where clients remain attached to their broker and have an application-level notion of location, which is described by *location-dependent subscriptions* introduced in Chapter 4 and put to use in Chapter 6. The location model specified in Chapter 4 is explicitly tailored to be directly deployable as part of the distributed routing infrastructure, on one hand, and to be usable for the convenient specification of location-dependent subscriptions in applications, on the other hand.

However, neither approach can prevent that applications in mobile and pervasive settings may face an inherent problem of asynchronous and event-driven systems: the need to listen to a stream of events until an application can commence operation from a consistent state. This is not a significant problem in conventional distributed systems settings, but in a mobile setting it can render an application useless because it may take too long to receive enough information for resuming operation. Hence, we devised novel mechanisms for placing distributed buffers in the network near the clients such that clients have convenient and timely access to information possibly delivered in the past and before the client's arrival at the current location. This effectively provides for decoupling not only in space but also in time of producers and consumers of information.

Finally, in Chapter 8 we define a framework for the development of context-aware applications, based on a structured approach leveraging finite state machines as a convenient abstraction for the

specification of an application's behavior. The notion of *Context* is used as input for the transition between the states. Additionally, we clearly show how such rather high-level information can be obtained from raw-data in the system by means of interpretation and aggregation of notification data.

Summing up, this thesis provides novel solutions to the challenges inherent in middleware support for pervasive computing environments. It clearly shows that leveraging the successfully deployed publish/subscribe paradigm is a convenient as well as appropriate way to address these issues. The solutions provided solve many problems of evolving the publish/subscribe paradigm to an important part of a middleware for supporting context-aware applications in mobile and pervasive computing systems.

1.3 Organization of this Thesis

The organization of this thesis is as follows:

- Chapter 2 presents the basic motivation underlying this thesis by giving a vision of future mobile scenarios and describes the setting in which subsequent solutions are located. It follows a characterization of the challenges infrastructural support faces in such highly mobile and dynamic settings. This is followed by a thourough analysis of the requirements for building appropriate support into the middleware. The central result is a number of requirements that are taken into account in the subsequent chapters.

- Chapter 3 lays the foundations of publish/subscribe systems for readers not familiar with this important paradigm as well as clearly defines the core properties of the assumed model of interaction. This is followed by the introduction of REBECA, the publish/subscribe notification service we used as starting point for our extensions. The chapter is concluded by a detailed description and discussion of related work to the contributions in this thesis.

- Chapter 4 introduces *location* as a source of information being of outstanding importance among data about the context. In this chapter we characterize the properties of *location* and location models and introduce an optimized reference location model for the use within a distributed publish/subscribe notification service.

- Chapter 5 is concerned with the integration of mobile devices into a distributed publish/subscribe notification servive. It characterizes the needs of applications running on such devices for transparent location handling within the infrastructure. A solution is presented that optimally takes into account such needs.

- Chapter 6 applies the location model introduced in Chapter 4 to scenarios where clients of the publish/subscribe notification service are *mobility-aware*, i.e., have to express their need for location-dependent information. A solution is presented that takes into account and facilitates for location-dependent subscriptions and notification delivery, on one hand, but also exploits the unavoidable uncertainty of message delivery for routing optimizations in the infrastructure, on the other.

- Chapter 7 introduces a distributed scheme of buffering in the broker network for appropriatly taking care of decoupling producers and consumers in space and time. The underlying observation is that mobile clients need a certain number of events to commence operation in

a new environment. Due to the asynchronous nature of event-driven systems, this can lead to unwanted delays or can even render applications useless. To cope with this situation we introduce the concept of *subscribing into the past*.

- Chapter 8 integrates the finding of the previous chapters into a model for the structured specification of context-aware applications for pervasive environments. Finite state machines are leveraged to specify behavior on the level of applications. Additionally, mappings are provided to transform specifications into a set of appropriate subscriptions for context-related data.

- Chapter 9 gives details on the implementation of the concepts presented in this thesis. We describe the implementation of the REBECA notification service which serves as a basis for the extensions made. After this we introduce implementation details of our mobility extension for accommodating mobile clients. These extensions are based on the algorithm introduced in Chapter 5. Then, details are given for the integration of location-dependent subscriptions into the core of the REBECA notification service as specified in the Chapters 4 and 6.

- Finally, Chapter 10 concludes this thesis by summing up the results and characterizing the directions of possible future work.

2 Ubiquitous and Pervasive Computing

"Ubiquitous computing is roughly the opposite of virtual reality. Where virtual reality puts people inside a computer-generated world, ubiquitous computing forces the computer to live out here in the world with people. Virtual reality is primarily a horse power problem; ubiquitous computing is a very difficult integration of human factors, computer science, engineering, and social sciences."

Mark Weiser

2.1 Introduction

The terms *ubiquitous computing* and *pervasive computing* were introduced quite recently. As a first impression of their meaning, we employ an official definition of *pervasive computing* provided by IBM:

> "*Convenient access, through a new class of appliances, to relevant information with the ability to easily take action on it when and where you need to.*" [HMNS01]

Mark Weiser [Wei95], the ultimate founder of ubiquitous computing, had a somewhat more holistic[1] approach in mind. He introduced the *vision* of ubiquitous computing (also called *ubicomp*) in 1991: he envisioned a future of "invisible" integration of computing hardware into everyday artifacts. In his fundamental article "The Computer for the 21st Century" [Wei91] he elaborated about "the computer that disappears". For Weiser the way into the 21st century was obvious and straightforward: technological advancements especially in the sector of micro-electronics will turn out as one of the major driving forces behind the possibility to implement and foster ubiquitous computing applications [Mat01]. Computers are getting smaller, cheaper, and more powerful. Networking technology, as a second major force, is following along the same path and will get smaller, cheaper, and more powerful, too. Consequently, as prices for computing power drops, it is expected that more and more everyday artifacts will be equipped with a reasonable amount of computing power and (maybe even more importantly) are networked together into a virtually unique network of communicating "*things that think*"[2]. Then, in the pure sense of the word, computing gets "ubiquitous": anywhere and anytime. This is considered to be the *third wave of computing*. In Figure 2.1 we characterized all three waves.

Ubicomp: the third wave of computing. *Mainframe computing* is the distinctive factor for the *first wave* as shown in the figure. It can be characterized as "*one computer, many people*": computing resources are scarce and hence are shared among a large number of users. Thus, usage must

[1] For a good definition of "holistic" see [Hol04] or the Merriam-Webster Online Dictionary[MW03].

[2] The "Things That Think" consortium is one of many projects that try to embed significant computing resources into everyday objects [Con03].

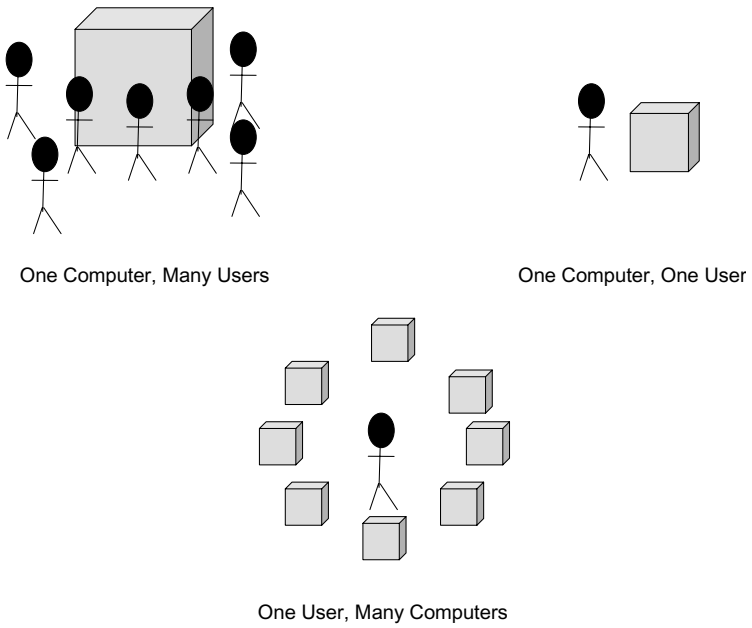


Figure 2.1: The three waves of computing

be well-prepared and planned ahead properly. Consequently, execution times were long. Computers were expensive equipment, run by experts behind closed doors.

The *second wave* is the current era of *personal computing* or "*one computer, one user*": since the mid-eighties the number of users of personal computers (PC) is larger than the number of mainframe users. People have direct control of and exclusive access to computing power. Tasks do not have to be planned well ahead anymore and interaction is spontaneous. But the drawback compared to the first wave is the responsibility for operating and maintaining such a system the user is burdened with. Especially if malfunctioning, considerable attention is required by the user.

The third wave is the upcoming era of *ubiquitous and pervasive computing* or "*one user, many computers*". As the first wave was characterized by every *computer* having many users, now every *user* interacts with many computers. Interaction with computers gets casual and subconscious. The user is the natural focus of computing and is moving through a world filled with (embedded) computers which are networked and interact between each other to spontaneously fullfil a user's demand. Everyone is continually interacting with hundreds or thousands of nearby computers. It is important to note that in such a scenario, not the computing power is the scarce resource (first wave), but the user's attention. Therefore, distraction intensive interaction (second wave) is not desirable.

Some people might consider such *pervasiveness* of computing and sensor hardware to be unsettling or even threatening. For Weiser, this technological trend leads to a simple thesis:

> "*The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it.*"

To explain this thesis, Weiser picks up the "technology" of writing as an example:

> "*Consider writing, perhaps the first information technology: The ability to capture a symbolic representation of a spoken language for long-term usage freed information from the limits of individual memory. Today this technology is ubiquitous in industrialized countries. Not only do books, magazines and newspapers convey written information, but so do street signs, billboards, shop signs and even graffiti. Candy wrappers are covered in writing. The constant background presence of these products of "literacy technology" does not require active attention, but the information to be conveyed is ready for use at a glance. It is difficult to imagine modern life otherwise.*"

Further he states:

> "*Such a disappearance is a fundamental consequence not of technology, but of human psychology. Whenever people learn something sufficiently well, they cease to be aware of it. When you look at a street sign, for example, you absorb its information without consciously performing the act of reading.*"

The challenge: making technology "calm". Building a bridge to computer technology, Weiser states in his article "Ubiquitous Computing" [Wei93] in 1993 :

> "[. . .] *The computer today is isolated and isolating from the overall situation, and fails to get out of the way of the work. In other words, rather than being a tool through which we work, and so which disappears from our awareness, the computer too often remains the focus of attention.*"

The essence of his believe was that we will have to rely on computers where ever we are and what ever we do. But, contrary to what we are used to, the *focus of attention* is different: today, computers are distracting and designed to attrack our attention; in his vision, computers are embedded in everything and simply "doing their job".

Given that the number of computing devices is expected to explode and to increase by a factor of thousand or even more, *unobtrusiveness*, or "calm technology" as Weiser put it [WB96], is a mere necessity. As striking as Weiser's vision is, as complex it appears to be realised. Although technology advances at a quick pace, nevertheless, it seems unlikely that it will reach the necessary sophistication and market quality within the next decade at least.

Pervasive Computing. The industrial term *pervasive computing*, introduced by IBM on their home page for pervasive computing [IBM01b], has a somewhat different focus. It does not stress the idea of "calmness" and "unobtrusiveness" as much as Weiser did. It follows a more pragmatic approach, in which the focus lies on *enabling technologies* for users to access data anytime, anywhere, according to their actual need.

The underlying thesis is that it is likely that some sort of "smart" device will prevail for a user's interaction with the surrounding physical and social environment. The thesis is that a personalized, implicitly trusted, mobile wireless communication device for interaction will be favored over a completely decentralized ad-hoc interaction model, as favored in ubiquitous computing.

Thus, *mobility* and *communication* in their various forms and with their implications for system design are central challenges. In this thesis we assume the idea of *pervasive computing* when formulating our system model in Section 2.2.1.

After this short introduction we will highlight some of the challenges this exciting new field of research imposes on "traditional" fields of computer science. The next Section will discuss the challenges for research on pervasive computing in more detail and analyze the additional requirements we have to face. To do so, we distinguish between requirements stemming from various levels of system design. In the center of the discussion we place *infrastructural support for a nomadic environment* as facilitator and mediator for mobile devices, mobile applications, and the physical surrounding, where computional resources are embedded and serve as partners for interaction.

2.2 Challenges in Pervasive Computing

The core challenge in pervasive computing, implicit in all the issues named so far, is what Lou Gerstner, past IBM Chairman and CEO, pinpointed in the following quote:

> "[...] *a billion people interacting with a million e-businesses with a trillion intelligent devices interconnected* [...]"

Although we do not favor the emphasis on "e-businesses", the challenge remains: to cope with large-scale pervasive computing systems where a large number of devices and software components is interacting. Interestingly, a pervasive environment can be large scale along two dimensions: (i) in terms of physical extension and (ii) in terms of "computers per cubic centimeter". While (i) is corresponding to a traditional model of large scale, (ii) is a consequence of the "third wave of computing" (cf. Figure 2.1 on Page 8). There, one assumes the surrounding to be saturated with computing devices, hence resulting in a large scale system which does not necessarily has a large extension.

Another approach and a more infrastructure-based characterization is given by [Sat01]: there, the challenges in pervasive computing are characterized along three stages (cf. Figure 2.2 on Page 11). The characterization of the problem space starts with issues raised by *distributed systems*, like, remote communication and remote information access. The next dimension is *mobile computing*, adding genuine new problems to system design, e.g., mobility and networking, as well as making the already known problems more difficult, like remote information access for *roaming* devices. The last dimension is *pervasive computing*, again introducing new challenges and making known problems more difficult, e.g., scalability. Prominent examples for genuine new problems are *smartness* and *context-awareness* , i.e., the ability of an application to adapt to local context and thus, to appear *smart*.

Kindberg and Fox [KF02] emphasize the influence of *physical integration* and *spontaneous interoperation*:

- *Physical integration.* In pervasive systems, the distinction between a passive object in physical space and computing nodes is softend. Physical objects can serve some special purpose in the physical world and be a computing node *at the same time*. For example, a SmartBoard [CTB+95; CFBS97] serves an actual physical purpose while recording what is written on it. Even a coffee cup might be *smart*, i.e., being equipped with computational and communication resources [BGS01]. Thus, integration between computing nodes and the physical world is necessary. This will be detailed in Section 2.2.2.

- *Spontaneous interoperation.* In the field of pervasive systems, the collaboration of *software components* builds up functionality such as services, clients, resources, or even whole applications. Contrary to a "classical" system model, this might be done on-the-fly. Hence,

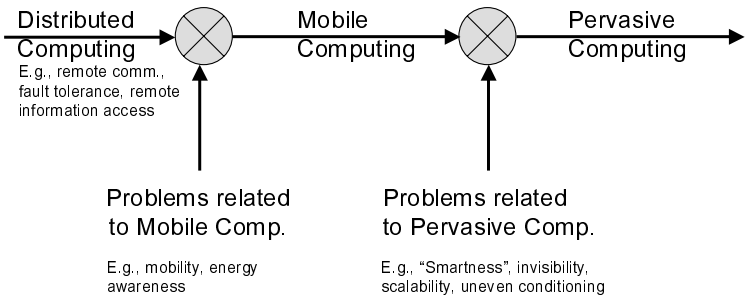


Figure 2.2: Growth of the problem space in pervasive computing

infrastructural components vary over time and space. Some entities might remain rather constant and change slowly, other might appear or vanish spontaneously. In [KF02], spontaneous interoperation is defined as the *ability to interact with a set of communicating components that can change both identity and functionality over time as its circumstances change*. Therefore, pervasive systems must be designed under the assumption that the whole system is highly dynamic and bindings are volatile. This will be detailed in Section 2.2.4.

Franklin [Fra00] summarized the envisioned challenges from the point of view of *ubiquitous data management*. Three key challenges are identified:

- *Support for Mobility.* Compactness of devices together with wireless communication means that devices can be used in mobile situations. Thus, legacy applications must be supported in a mobility transparent fashion as well as new applications that are built location-aware.
- *Context Awareness.* This includes the support for environment-aware devices, which want to make use of environment-related context. Context-aware applications range from "intelligent notification systems" that inform the user about events or data, to "smart spaces", places/environments that adapt to the users.
- *Support for Collaboration.* Support is needed for the collaboration of groups of people or devices.

In Lee et al. [LXZL02], the high-level challenges for building location-dependent applications and services, as the most significant class of context-aware applications, are summarized as follows:

- *Mobile environment constraints.* Mobile-pervasive environments are resource limited in terms of bandwidth, quality of communication, frequent network disconnects, and limited local resources.
- *Spatial data.* Data provision can be dependent on the current location of a user or device. Moreover, relevance and validity of data might vary according to the location.
- *User movement.* Common tasks for data management, e.g., caching, become complicated because of roaming users.

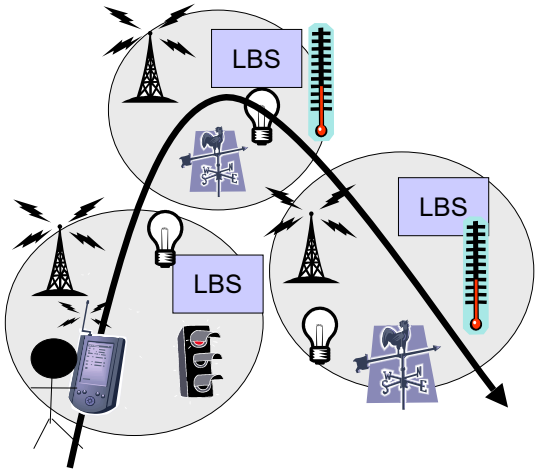


Figure 2.3: Nomadic computing model

We believe that such devices will depend on a strong infrastructure supporting interaction and offering means for data management. "Trillion devices" will have to rely on strong mechanisms to *regulate* the flow of information produced by other devices or software. Otherwise, *relevant* information will be "drowned" in the "noise" produced by other entities. Often, sensors produce data periodically or information needed by other entities for interaction is floating through the system. Thus, the naïve mode of simply *flooding* data through the network does not scale to the systems we envision.

After this overview, throughout the next sections we will analyze the requirements for a middleware platform from the perspective of the participants of such a system. But first we want to clarify the actual system- and application model, which we assume as foundation for the remainder of this thesis.

2.2.1 System Model

As we have shown, the terms ubiquitous and pervasive computing span a whole range of different application domains and therefore system models. However, we have to limit this to a concrete instance of a system model. In Figure 2.3, we have sketched the model we adhere to in the remainder of this thesis: the *nomadic* model of pervasive environment, as it is also used in the cooltown project [KB; KBM$^+$].

The model presented here is based on certain assumptions about the hardware and communication resources available:

- *Nomadicity.* In the center of the model are the users who will be mobile and roaming around.
- *Handy devices.* Nomadic users will carry and use some mobile devices as *enabling technology* for accessing the surroundings. These devices usually are personalized and have reasonable

resources.

- *Wireless communication.* Easy-to-use wireless communication is expected to be built-in.
- *Heterogeneity.* There will be a diversity of devices. Specialization of function, e.g., digital cameras and phones, will produce heterogeneity. Moreover, sensors and actuators can be expected to be part of the surroundings.
- *Fixed resources.* The infrastructure has to provide fixed services, like printing, teller machines, and kiosks. Support for location-based services can be expected.
- *Wired backbone.* The user is moving in an infrastructure where fixed resources also imply strong networking and reliability.

The requirements we introduce throughout the following sections are based on this model.

2.2.2 Nomadicity and Devices

Kindberg and Fox, [KF02], among others, identify the need for integration of "smart things" into a computing infrastructure. Besides this integration, the nomadicity of users and their devices raises additional issues.

Devices can be mobile for a number of reasons, usually because they are carried around by a user, like a mobile phone, a PDA, or a wearable computer. In other settings they might be mobile autonomously, like robots, or because they are embedded into some larger device, like a car or a bus.

In the nomadic computing model, devices are mobile in changing surroundings, each full of other devices, placed within the environment so that they become pervasive (cf. Figure 2.3). Ubiquitous computing, on the other hand, originally has focussed on "backgrounding" devices, disappearing into the environment. In this thesis, we take the general standpoint that devices are not "fading" into the environment and are "magically" doing the right thing, but that devices are explicitly available for collaboration and interaction. The "quality of interaction" to be expected from a device strongly depends on whether the device is itself mobile. A device embedded into the infrastructure usually has other resources available than a device that is carried around.

2.2.2.1 A Taxonomy of Device Mobility

In this section, we roughly follow the approach taken by Dix et al [DRD+00]. The first destinction to make is the *type of mobility* we can assign to a device:

- *Fixed.* A fixed device is not changing location as in the mobile case. This class includes sensors spread within the environment or base stations fixed at a particular place[3].
- *Roaming.* A roaming device is moving around and usually is subject to the will of others, like a PDA carried around by a user.
- *Autonomous.* A device which is moving under its own control, e.g., a robot.

The next distinction is the relation to other devices encountered when mobile:

[3] Please note that even fixed devices might change location over time. For example, when attached to a different part of the infrastructure for management reasons.

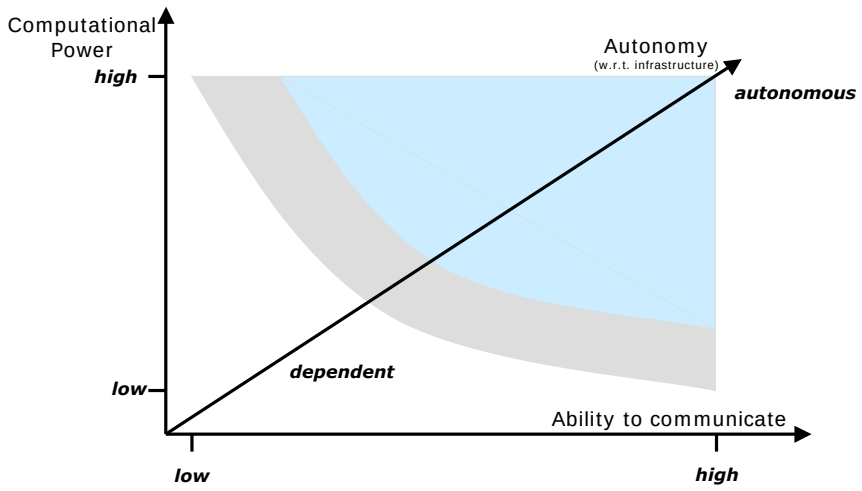


Figure 2.4: Design space for small devices

- *Free Roaming.* An important class of devices for this thesis are devices which are basically self-contained. Their functionality is mostly independent from other devices in the vicinity or the environment they operate in. Bindings into the infrastructure therefore are volatile and transient. Examples are PDAs and wearable computers that explicitly are designed to be independent of external sources. Interaction with other entities is opportunistic, often in order to extend their basic functionality, e.g., to act location-aware. We discuss this later in the context of *location-dependent subscriptions*. On the other hand, delegation of certain aspects of mobility into the infrastructure in order to be oblivious to mobility is also an important issue, and appropriate support by the infrastructure is needed.

- *Embedded and Pervasive.* The other extreme are embedded and pervasive devices. Often these are devices other "free roaming" devices interact with. In this class we place sensors, actuators and embedded devices, which are not self-contained by themselves, but are part of a larger system. This class of devices either is source of information, e.g., sensors in a pervasive environment, or rely on external funtionality provided by the infrastructure in order to operate properly. An example might be a location-based service on a PDA.

After analyzing mobility from the viewpoint of mobile devices, throughout the following subsection we analyze the need for infrastructural support for mobile devices, complementing the above stated.

2.2.2.2 Infrastructure Support for Mobile Devices

Mobile and stationary devices will be part of any pervasive computing system. Currently, only few devices offer the amount of computational resources required, e.g., for spontaneous networking using

Jini or SLP. Figure 2.4 [KVZ99] shows the design space of devices concerning their capabilities with respect to three different dimensions.

- *Computational Power.* With this first dimension we assess the computational or processing power a device has. This includes CPU performance, memory size, and size of persistant storage. They are subsumed into one single dimension as they usually grow at comparable rates as devices get larger. However, computational power is a gage for a device's capability to process data locally. Low computational power corresponds to the potential need for external preprocessing.

- *Communication.* In pervasive systems the ability to communicate and to network is an important factor and constitutes a separate dimension. This includes bandwidth, the underlying network technology used, and related issues, like online vs. offline operation or support for uni- vs. bi-directional communication. The ability to communicate is an indicator of the degree a device can participate in a pervasive environment. A device with low bandwidth should not receive too much information at a time, otherwise the result is congestion on the network link. It might also serve as an indicator for the necessity of pre-filtering information externally to avoid such congestions.

- *Autonomy.* The third dimension in Figure 2.4 is labeled *autonomy* and indicates the ability of a device to operate independently of the surrounding infrastructure. As autonomy is strongly influenced by the other two dimensions, it is not truly orthogonal. The shaded regions are meant to mark the relation between the first two dimensions and autonomy: the more powerful a device is along the first two dimensions, the more the device is likely to be able to operate without explicit support from the infrastructure. However, other factors besides processing power and communication can pose serious limitations to the autonomy in a pervasive environment. The most important are:

  - *Output.* The capability to which extent a device can interact with a user is limited by the available facilities for output. Limiting factors include display size, color vs. monochrome, or audio.
  - *Input.* Important for user interaction and therefore for autonomy is the efficiency of available input devices, like keypad, voice, pen input, or keyboard.
  - *Flexibility and Extensibility.* Autonomy is directly related to the generality of a device, i.e. the ability to dynamically adapt to changing environments. This includes the ability to install new software as well as the possibility to upgrade a device.

As illustration of the design space, we can identify certain device categories which we will revisit and take as guiding examples throughout the remainder of this thesis. The classification should give an impression of the character of such devices and the kind of support needed from the surrounding infrastructure, i.e., tasks to be done externally.

- *Sensors.* Probably, the most ubiquitous devices are sensors. They are embedded into the physical world and usually provide the most basic input for pervasive applications. Sensor devices typically are equipped with the minimal computing power required to fulfill their tasks. Communication facilities are wire-based or wireless, e.g., infrared. The typical purpose is to serve solely as producers of specialized raw data. Thus, communication is uni-directional and usually over a low-bandwidth link. Additionally, their mode of operation is often either

event-driven, triggered by external changes in the physical surrounding, or periodical, i.e., data is sent in short time intervalls, in contrast to a continuous transmission.

Examples include position sensors as used in the Active Badge (e.g., [WHFG92]) and ParcTab (e.g., [WSA+95]) systems or sensors for environmental parameters as used in the Adaptive Home project [MDA+; Moz98]. Processing of data is done externally in the infrastructure or in some applications, respectively (see also Chapter 8).

- *Actuators.* Complementary to sensors are actuators. They also are embedded into real-world objects, serving the opposite purpose than sensors. They are able to receive and process commands as well as data in order to manipulate some entity. Usually, they implement a protocol of more or less high complexity used to influence the behavior of the actual device or the actuators.

  To conserve energy, such devices are often intermittently offline. Hence, the infrastructure or some other entity, e.g., a device proxy [ADH+99], facilitates basic management and integration services on behalf of the device.

- *Information-processing devices.* This class of devices can be assumed to be residing in the lighter shaded area of Fig. 2.4. They have enough computational power to participate properly in a certain environment properly and perform most of the basic tasks on their own, like integration into the infrastructure. Nonetheless, even those devices might suffer significantly from a comparatively slow and bandwidth restricted wireless link or too much data sent to them directly for processing.

Integration of small devices. Devices, which are dependent according to the characterization of Figure 2.4, rely to some extent on support from the infrastructure. One might argue that devices belonging to this category will eventually become powerful enough to operate autonomously and the problem of integrating small devices will vanish by itself, an assumption met by many modern electronic artifacts. However, we argue that with more devices integrated into a system the desire to integrate additional and smaller devices will grow. Hence, we state the following prediction on the integration of small devices into mobile, pervasive environments:

REQUIREMENT 2.2.1 **Middleware support.** *There will always be a significant class of devices which is dependent on the support from the surrounding infrastructure in terms of computational power, memory, persistent storage, or facilitation of integration.*

2.2.3 Applications

The outstanding common denominator for pervasive applications in comparison to mobile applications is that such applications typically try to make use of the current surrounding. In a sense they are *aware* of their *context*. Sometimes, the term *Context-Aware Computing* is used in a sense almost synonymous to ubiquitous computing, emphasizing the importance of *context*. For example, in the Active Badge project an application named *teleporter* is responsible for displaying data on a display near a user, taking into account the location (context) of that user. In tourist guides, information is displayed relative to a user's position and orientation. An augmented meeting room might forward calls relative to the *situation* in the meeting room. All these are examples for context-dependent behavior of applications. The ultimate goal for context-aware applications is to *adapt* their functions according to the current situation "anytime and anyplace".

2.2.3.1 Context and Context-dependent Applications

Using *context* is not a new concept for Computer Science. *Context* is often used as an umbrella term for all external data influencing the execution of adaptive and reactive applications. The most prominent examples for adaptive behavior in applications can be found in Artificial Intelligence and Human-Computer Interaction. In the latter case, the provision of a customized user interface is a common research topic. In Artificial Intelligence the issue of context arises in assorted areas, like knowledge representation, natural language processing, or intelligent information retrieval. An example for the early use of context can be found in McCarthy's [Pag84] criticism of MYCIN [Sho76], an expert system for advising physicians on certain infections. As we consider issues of artificial intelligence out of scope for this thesis, we refer to, e.g., Akman and Surav [AS96] for an extensive summary of *context* in AI.

Our concern in this thesis is *context-awareness* as incarnation of adaptive behavior for the area of mobile communication, which we consider to be a main driving force for the development of mobile, pervasive applications.

Dey, et al. [DA00; DSA01], provide a rather good overview over common definitions of the term context. Another extensive survey can be found in [CK00].

Interestingly, a common observation in the literature, e.g., [DSA01; Win01], is that although the term context is intuively understandable it is poorly defined. Therefore, comparison of research related to the use of *context* is hard to conduct. The definition and use of context is problem-centric and iterative rather than universal and normative, the latter more desirable for the definition of conceptual models as the one detailed in this thesis.

Nonetheless, we want to start by discussing some of the most common definitions as they are used in the field of pervasive computing. One of the first documented uses of the term *context-aware computing* can be found in Schilit et al [SAW94]. As many other authors after them, Schilit et al., classify a new class of (mobile) applications "*that are aware of the context in which they are run*". Such "*context-aware systems*" adapt according to the location, collection of nearby persons or devices, and the *changes* of such collections over time. Thereby, they already emphasize the importance to react to events occuring over time in the physical world around an application. The same point of view can be found in [DCEF02], where the authors state: "*the defining feature of mobile environments is the concept of change.*"

This is leading directly to the requirement to support such reactive behavior:

REQUIREMENT 2.2.2 **Reactive Behavior**
Context-dependent applications require appropriate support for reactive behavior as they have to respond accordingly to events occuring in the surrounding physical space by adapting their thread of execution.

Enumerative and operational definitions. Common problem-centric definitions simply enumerate classes of physical raw data that can contribute to an application's thread of execution. In [BBC97] context consists of location, identities of people, time, season, temperature, etc. Similar, in [PRM99], context is defined as location, environment, identity and time. These definitions are difficult to apply since they only describe instances but not the underlying class of context, making it hard to decide whether potentially new types of context information can be classified as context information or not. Other definitions have provided synonyms for context, such as *environment* [Bro96; WJH97] or *situation* [FJ98; HNBR97], that are used to emphazise certain aspects of the more general context concept. A widely adopted operational definition is given by Dey and

Abowd in [DA00]. They define context as "*any information that can be used to characterize the situation of entities [...] that are considered relevant to the interaction between a user and an application, including the user and the application themselves. Context is typically the location, identity and state of people, groups and computational and physical objects.*" However, as it is also remarked by Winograd [Win01], the use of open-ended phrases such as "any information" and "characterize" is convenient for covering specific work of the authors but does not help to define context in general. An operational definition should provide means to discern context information from information that is not regarded to contribute to context adaptation. Such helpful and operational definition can be found in [Win01] and characterizes the use of context in communication: "*context is an operational term: something is context because of the way it is used in interpretation, not due to its inherent properties.*" In other words, if something or someone is acting dependent on input of particular data, then it is context information.

System-oriented definitions. Another view on the "*context-problem*" is provided by Lieberman and Selker [LS00]. They postulate that context is "*beyond the 'black box'*". For them, the field of computer science is strongly bound to a position which is "*antithetical to the context problem: the search for context-independence*". Many areas of computer science explicitly treat the systems of interest as black boxes. Especially, in research on distributed and networked systems, the ultimate goal is the *transparency* and *independence of* underlying technologies. A rather good example for this observation is the specification and semantics of MobileIP [Per98a; Per98b]. There, mobility is completely opaque to an application and resulting in transparency of mobility. In Harter, et al., [HHS+99] and [HHS+02], a similar need for knowledge about "*capabilities of the equipment and networking infrastructure*" is expressed. A similar view can be found in [GDH+01]. They emphasize the danger of *transparent* access to remote resources in the face of network and remote resource failures.

In traditional systems, an output function is completely determined by the (direct) input provided. In [LS00], the authors stress the need to broaden the notion of input to the use of *context* as a necessary source of input parameters. Interestingly, they identify a trade-off between the desire for abstraction, as a means of determinism wanted, and context-sensitivity, as a means of provide adaptation and flexibility. Consequently, the extent to which a system is context-aware then depends on where the boundaries of the system are. This system-centric view is different from most other definitions that follow a strict context-centric point of view.

REQUIREMENT 2.2.3 **Beyond the black box**
For context-aware applications knowledge about underlying system technologies can be an important source of information. Hence, a pure black-box model is not applicable anymore. Certain features of the underlying system structure should be accessible to an application.

2.2.3.2 General Context Models

In this section we will try to identify different categories of *context* by exploring given classifications found in the literature. The focus thereby lies on classifications which are helpful for computer-based mechanisms.

As one of the first coarse characterizations, Schilit et al. [SAW94], classify context along the categories:

- where a user is;

- who a user is with;
- what resources are nearby.

This characterization includes more than the location, as "*other things are also mobile and changing*." However, this characterization basically is "flat" and unstructured. Other authors proposed a hierarchical model of context, often using first level, second level, and so forth, context classifications. One example for such a model can be found in Dey and Abowd [DA00] and Abowd and Mynatt [AM00], where first-level context is classified by the questions *who, what, where, and when*, pointing to next-level context, like the email address of a user. Along the same line of thought Lieberman and Selker [LS00] define three distinct models of context, a user, task, and system model. The first is a description of the user, the second a description of the goals and actions to achieve, and the last a model of the system in which a device or application operates.

In the cooltown project [KBM+] another classification is used:

- *People.* The people abstraction has relationships to all context related to a user. This includes people in space, other users, etc. In cooltown the first-level context data is a URL to the so-called *web presence* of a user as the first-class context and pointer to next-level context.
- *Places.* Places represent the physical surrounding and is a portal to more context information (cf. *Things*). Places also own a web presence, managed by a central place manager.
- *Things.* Any physical object that can be relevant to an application. Usually, things are associated to places and have a web presence, too.

A more object-oriented approach is found in Schmidt, et al. [SBG99] in order to develop a hierarchical *feature space* of context. They propose the following model:

- A context describes a situation and the environment a device or user is in.
- A context is identified by a unique name.
- For each context a set of features is relevant.
- For each relevant feature a range of values is determined (implicit or explicit) by the context.

At top level they place context related to human factors in the widest sense, and context related to the physical environment. Based on these categories they distinguish them further into three subcategories each (user, social environment, task; and conditions, infrastructure, and location, respectively). Those are then used as a basis and general structure for context. Additional context is assigned to one of those six basic categories.

Dix, et al [DRD+00], pick up a similar approach and state the role of *relationships* for context, as a common denominator for classes of context. They distinguish between *infrastructure, system, domain*, and *physical* context as the different categories (cf. Table 2.1). Being similar to the classifications mentioned before, Dix, et al., heavily emphazise the importance of *location* as an "*indexing device*" from which to infer the overall context and influencing all other context categories. They state that the very idea of mobility alone demands an understanding of *location as concept*. Furthermore, it might be used to be exploited as a means for a global understanding of the system. The importance of *location* as a rich source of information is an agreed-on issue in research on context-aware computing, e.g., [AAH+97; DCME01].

context	*relationship with*	*Issues*
infrastructure	network bandwidth, and reliability, display resolution	variability of service, user awareness of service, liveness of data
system	other devices, applications, and users	distributed applications, pace of feedback and feedthrough, emergent behaviour
domain	application domain, style of use, identification of users	situated interactions, personalization, task and work studies, privacy
physical	physical nature of devices, environment, location	nature of mobility, location dependent information, use of environmental sensors

Table 2.1: Categories of Context in Dix, et al.

REQUIREMENT 2.2.4 **Location as first class concept**
Because of the outstanding richness of location as concept, we require that location *should be a first class concept in the specification and implementation of context-aware applications.*

2.2.3.3 Programming Models for Context-aware applications

The definitions of *context*, as introduced above, show that in theory every piece of information may possibly contribute to the execution of context-aware applications. However, due to the nature of mobile, context-sensitive applications, applications have to be specified separately and independently from a specific system setting. An application that should be useful in changing environments cannot make explicit assumptions about the environment it operates in. This constitutes the need for adaptive behavior on the application level. Direct access and control over resources in the environment usually is not feasible or at least hard to implement. In an environment where multiple separate applications want access to the same information, producers should be decoupled from the actual consumers of data. This measure introduces the necessary anonymization of producers and consumers. Sometimes this is done for security reasons, but more often to make resources sharable among a large number of clients. Thereby, data is produced independently of knowledge about the consumer, therefore adding flexibility to the system.

Unfortunately, from the application's point of view, the problem of data aquisition becomes important. An application can hardly make concrete assumptions about the actual instance of a data source. This includes the acquision frequency and time, the mode of disemmination of data items, and what syntax and semantics the data exhibits. Consequently, a programming model for context-dependent applications should distinguish between the specification at *designtime* and the behavior at *runtime* (cf. also [BBG+00]).

Designtime. As direct control over resources and concrete data is not possible, a designer has to resort to a data-oriented approach and specify an application in terms of *reactive* behavior. At

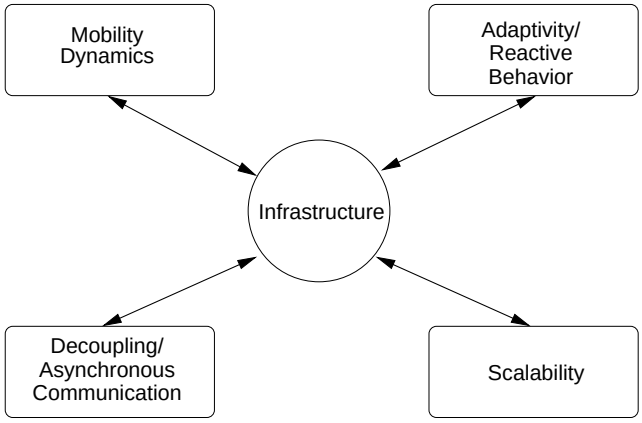


Figure 2.5: Requirements for infrastructure support

design time the general behavior of an application is described as *tasks* or *goals to reach*. For an application it is necessary to react to *what* happened, rather than *how* this information was acquired in the first place. We require the specification thus to be on a high level of semantics. For instance, a "conference room" application has to react to the *presence* of people attending a conference, independently of whether this information was acquired by a video-camera, an RFID tag reader, or an IR-badge. In this respect, we require a reasonable separation of goal-oriented specification and the actual instantiation at runtime.

Runtime. At runtime, the semantic specification and reactive behavior of an application, as proposed above, must be implemented through data-acquisition and aggregation of data from available resources in the surrounding. Therefore, monitoring and distribution of events must be supported by the environment and appropriate means must be provided for applications to filter and aggregate information suitable for direct input as contextual information at runtime. Obviously, part of this functionality can be externalised and should be located in the surrounding infrastructure for efficiency and flexibility.

Usually, the definition of programming models is problem-centric and "ad-hoc". Therefore, Chapter 8 will tackle the problem of modeling applications in more detail and provides a state pattern oriented approach to the specification and implementation of context-sensitive applications.

2.2.4 Infrastructure Support for Pervasive Environments

In the previous section we have already described some of the requirements on an infrastructure for pervasive systems. Figure 2.5 summarizes these requirements we want to focus on in the course of this thesis. Thus, this section is devoted to an analysis of requirements from the viewpoint of infrastructural support. The goal is to identify areas of middleware support where "traditional" infrastructure for distributed systems has to be extended for use in the nomadic computing model.

2.2.4.1 General Problem Statement

In static distributed settings, a middleware usually deals with fixed devices and rich computational resources. For networking one assumes permanent connections and a static context, i.e., the overall settings and relationships between the participants of the system are changing only slowly. Naturally, the rationale for the deployment of *middleware* is to mask the unwanted aspects of *distribution* of computation and act transparently. The ultimate goal for a middleware to achieve in such settings is to appear as perfect *black box*: it is used via well-known and well-defined interfaces and transparently hides all internals from the application layer. A "virtual single processor" model thereby is extended to the distributed case. However, given the strong guarantees above, i.e., static environments and reliable networking, the model of a black box is a valid approach. Even complete parts of the application logic can permanently be relocated into the middleware.

Although existing middleware, like transaction-oriented, message-oriented, or object-oriented middleware (cf., e.g., [Emm00; Ber96]), is successfully deployed in static environments, it is likely to fail in mobile settings. In a nomadic computing system, transparency of the infrastructure and its distribution is only one aspect and not even always wanted. As we have shown above, explicit awareness of the environment is a rich source for adaptive behavior and can be exploited to various degrees for the benefit of an application or a user. To support nomadic, mobile systems we have to "open-up" the black box. Moreover, most of the guarantees given for static distributed systems are unlikely in mobile environments, as we shall show below.

In [Gei01] Geihs states that the requirements for infrastructure support "*changed dramatically since the era in which early middleware developers worked in environments dominated by locally connected Unix workstations.*"

2.2.4.2 Characterization of Infrastructure in Static Environments

A useful general definition for middleware or infrastructure is taken from [HL01]: "*An infrastructure is a well-established, pervasive, reliable, and publicly accessable set of technologies that act as foundation for other systems.*"

For the definition of middleware services, we extend the definition given in [Ber96]:

DEFINITION 2.2.1 **Middleware Services.** *A middleware service is a general purpose service that sits between platforms and applications. Platform means some computational architecture, i.e., processor and operating system. A middleware service is defined by the APIs and protocols it supports and services a large class of applications.*

In distributed, non-mobile systems, middleware usually is characterized by:

- Fixed devices, e.g., servers with rich resources for heavy workloads and desktop PCs with sufficient resources as clients, following an implicit client/server paradigm.
- Permanent connectivity, with high bandwidth and reliable connections. Communication usually is done synchroneously and tightly coupled, e.g., by *Remote Procedure Calls* [Sun88; Gro97] or the more recent object-oriented variation of Sun's *Remote Method Invocation* [Mic97].
- Static context, where the overall system evolution is slow, middleware can therefore implement application knowledge.
- Transparency, i.e., aspects of distribution and functionality are hidden in the infrastructure (*black box model*). Interaction is *solely* defined by the interfaces and protocols supported.

2.2.4.3 Requirements for the Infrastructure Support in Mobile and Pervasive Systems

The first observation on infrastructure for mobile and pervasive systems is that *mobility support* is an inherent requirement. But, mobility breaks many assumptions cultivated delicately and made implicit in the design of middleware for static environments and their applications. Where traditional middleware tries to hide characteristics of the environment from an application, the very same information might be useful for a mobile application.

Many viable assumptions for a static system only hold to a certain extent. Therefore, the determinism of the overall system is restricted. To give an example, one distinctive characteristic in distributed system design is to make certain assumptions about failure modes. But, many failure modes of static systems are modes of *normal operation* in mobile systems. For instance, wireless networking naturally introduces slow or unreliable network links and battery-driven devices are switched off regularly to save energy. Both examples introduce change and unpredictability for communication. An infrastructure for nomadic environments *should* provide mechanisms to bridge such erratic behavior of clients. One possible solution is to introduce asynchronous communication as underlying communication paradigm. Thereby, we weaken the assumption of synchronous communication that sender and recipient have to be available at the same time.

After this short introduction and problem overview, we will detail the problems involved and move along the requirements as shown in Figure 2.5.

Mobility. For traditional middleware the provision of complete transparency of the underlying technology and the surrounding environment is the ultimate goal to achieve. Where traditional middleware simply hides mobility and change from an application, two distinct modes of operation are needed in mobile settings: *transparency* and *awareness* of mobility.

The infrastructure we envision must operate in highly dynamic environments where client mobility is the norm. Clients move around and appear or disappear as they come into or leave the reach of the current context. Support for such forms of nomadic computing environments is mandatory.

However, not all applications are explicitly targeted towards the use in mobile scenarios. For example, so-called “legacy applications”, like the famous “stock ticker”, are not necessarily aware of mobility when run on a mobile device. Here, we want the infrastructure to provide transparency of location and mobility. The infrastructure should handle the details of mobility-related issues, like bridging phases of disconnectedness [ZF03]. On the other hand, applications might want to take advantage of some sort of *awareness*, like device- and environment awareness [CEM01] or location-awareness. In Capra et al. [CEM01], *device awareness* is relative to the information available on a device, and *environment awareness* is relative to the environment in which computation takes place, respectively. The most prominent example for environment awareness is context-sensitivity in applications. Yau and Karim [YKW+02] define the task for a “middleware-oriented approach” as “striking a balance between awareness and transparency to the application.”

REQUIREMENT 2.2.5 **Transparency vs. Awareness**
A middleware should support transparency as well as awareness of mobility at the same time.

Dynamics, adaptivity and reactive behavior. By definition, any infrastructure should be extensible, must cope with evolution, and should provide means for openess. Although these requirements were the core reason for building middleware in the first place, they gain more importance in mobile and pervasive settings where the overall setting is highly dynamic and bindings between

clients and services are volatile and casual at most. In pervasive settings *flexibility* as an answer to high dynamics is of outstanding importance.

Another issue is the inherent heterogeneity of mobile environments. Applications cannot expect to have a uniform hardware or service platform to rely on. The task to locate, associate, and coordinate activities of entities on-the-fly needs support by an intermediary infrastructure. Additionally, the description of functionality should be independent from the actual technology found in a certain environment in order to separate the semantics of functionality from the actual implementation and the technology applied [DCEF02; HL01]. This provides means for evolution and extensibility. For example, consider an application bound to a single location model and sensing technology. When the technology changes over time, and therefore possibly the location model, the application might be rendered useless. In such situations, a middleware providing for abstractions from the technology used is beneficial.

As a requirement, we can state:

REQUIREMENT 2.2.6 **Extensibility and Flexibility**
Middleware must be usable as intermediary between applications and environments in terms of extensibility and flexibility.

Adaptation [KF02; YK03] is the second prime requirement agreed on in the literature and a direct concern identified above in Section 2.2.3.

Applications that are mobility-aware and context-sensitive need support for adaptation to a new situation in a new environment. Be it as "flexibility to integrate new sources" [DCEF02], "obtaining and discovery of information" [HL01], "abstraction of context acquisition" [YK03], "programming for change"[GDH$^+$01], or "task dynamism" [BB02], support for various different tasks is needed. Another form of adaptation obviously is coping with fluctuations of resources, or more general, "dynamically varying resource supply" [Gei01]. Hence, we require

REQUIREMENT 2.2.7 **Adaptation**
The infrastruture has to offer means for the adaptation of applications to new environments.

An important implication of the above requirement is that the whole procedure of application development must be adapted to match the need for adaptive behavior on the application level. Middleware support therefore includes the "implementation" of adaptive behavior at runtime. Besides the efficient delivery of notifications about changes, mechanisms at runtime are needed to complement the programming model used for the specification of applications at designtime. However, applications are specified in a reactive, process-oriented fashion, i.e., in terms of (simplified) "`whenever` $< x >$ `occurs`, `do` $< y >$". The underlying data-model of the nomadic system should reflect such specifications, where the "what", i.e., the actual data item, is more important than the "who" has produced the data. This data-centric view is shared by, e.g., Kindberg and Fox [KF02]. Consequently, the specification of data-queries should be data- and event-oriented.

Decoupling in space and time. For static distributed systems the request/reply paradigm has been the predominant programming paradigm. As we have shown, in the nomadic computing model many assumptions have to be redefined and, consequently, other programming models than blocking, synchronous pull-based interaction is needed. For a detailed discussion on this issue refer to [FMB01].

The main reason why the synchronous request/reply paradigm is not sufficient anymore, is the underlying assumption that: (i) client and server are "known" to each other, i.e., addressable directly, and (ii) are available for communication at the same time. In other words, synchronous communication needs tight coupling in space and time to function properly.

As already mentioned above such conditions are hard to maintain in mobile environments with fluctuating resources, e.g., in wireless sensor networks [RKM02]. Devices might be powered off to save battery or are unreachable due to a link breakdown. The infrastructure in this respect has to serve as the intermediary between producers of data and consumers and provide decoupling in space and time.

Dix et al. [DRD+00] observe that connectivity is better characterized as "intermittent connectivity", rather than "intermittent *dis*-connectivity," resulting in the need for asynchronous and decoupled communication. The same argumentation is pursuited by Kindberg and Fox [KF02], who argue that disconnection is no fault, but at most a transient failure. Therefore, any system has to be built *robust* enough to cope with "routine failures", such as a powered-off device that is not available. In Dix et al. [DRD+00], it is argued that a middleware has to facilitate the distributed handling of the "event phenomena", i.e., communication is inherently event driven, like when communicating state-changes. As another positive result of decoupling sender and receiver of data, Hong and Landay [HL01] argue that sensors, services, and devices, as participants of the global system can be changed independently, dynamically and at runtime, thereby contributing to the capability to adapt to evolving systems.

Winograd supports the need for decoupled operation in the field of context management. The underlying assumption is the higher the grade of decomposition of a system, the greater is the need for decoupled operation (cf. also [Win01]). He illustrates this by examining three common "middleware" abstractions for pervasive systems: widgets, networked services, and blackboards.

- *Widgets.* The notion of widgets is tightly bound to the work of Dey, et al., [SDA99; DSA01] on the Context Toolkit. This approach originally stems from the development of user interfaces where widgets hide the specific details of some hardware device driver and present a unified interface. The widget model grew from a tradition of tight-coupling and single-manager control [Win01]. Only recently was a layer of abstraction introduced, the so-called discoverer, that allows for a slightly more decoupled mode of operation, where relationships between entities can be set-up at runtime, thus building a bridge to the networked service model. However, the underlying interaction paradigm of the Context Toolkit is based on the client/server paradigm. For example, sensor widgets act as context servers and applications that request data from widgets are context clients.

- *Networked services.* Networked services are systems centered around a service paradigm. Clients have to find the location of a certain service through pre-configuration or a discovery process, like it is used in Jini [Sun99b], and then set up a transient or permanent connection to this service. However, compared to the rather static setting of the widget model, components expose a greater level of independence. An example for such an architecture is presented in [HL01].

- *Blackboard.* Blackboards, as they are proposed in [Win01; FJHW00], use a paradigm of a centralized blackboard, realized through a tuple space, through which producers and consumers of data coordinate their interaction in a de-coupled and data-centric fashion. The authors assess this mode of interaction as being more suitable for pervasive systems than the others

characterized above. On the other hand, they neglect the inherent negative properties of such a centralized model, like limited scalability.

Using a data-centric and decoupled infrastructure model obviously provides the greatest freedom for interoperation of the various components building up a system. Another communication paradigm providing for asynchronous and decoupled communication is the publish/subscribe paradigm. In the context of pervasive computing it is exploited to some extent, e.g., in the Nexus project [NM01; Til02] and the Portolano project [EHAB99; GDL+01]. We will analyze the suitability of the publish/subscribe paradigm in greater detail in Chapter 3 of this thesis.

As a summary of the discussion above we introduce the next global requirement:

REQUIREMENT 2.2.8 **Decoupling in Space and Time**
Inherently, large-scale mobile and pervasive systems require support for decoupled and asynchronous interaction and communication in space and time.

Scalability. Given the sheer number of sensors, services, and devices envisioned to comprise the future of ubiquitous computing, inherent scalability is an important issue for ubiquitous computing middleware. Infrastructures need to work for large numbers of "everything" [BB02], as the distinction between computing device and common physical artifact is diminishing (cf. Sect 2.2 and [KF02]). Hence, the scalability of a system is related to the ability of serving larger numbers of mobile devices in an efficient way [MCE02].

In Kindberg and Fox [KF02] a "boundary principle" is introduced. Scalability is a concern of two levels: (i) within a single boundary, i.e., along with the number of participants within a domain, and (ii) across boundaries, i.e., the federation of a number of such smaller systems into a larger system where interaction takes place across each system's boundary.

REQUIREMENT 2.2.9 **Scalability**
A middleware must inherently be built for scalability.

Support for small devices. In accordance to our own analysis in Section 2.2.2.2, the extensive support of small devices is a requirement widely accepted for a pervasive environment. While [KF02] talk of heterogeneity of devices, especially in embedded systems and the therefore increased need to provide a uniform layer of abstraction, [BB02] take into account the resource constraints heterogenous devices might have and require middleware to provide additional help. In [LS00] the authors explicitly state the problem of devices which are easily overwhelmed with the sensory data available and thus the demand to have some pre-filtering of data elsewhere. Even more extensive support is advocated by [HL01], they postulate that as much as possible of the necessary processing is performed by the infrastructure rather than on a device. Another issue, although obvious at first sight, is that for the support of resource limited devices a common communication platform is favorable. Communicating devices then at least can rely on a set of conventions for interoperability. This becomes increasingly important in the face of a growing number of heterogeneous platforms and data formats [GDH+01; GB03; KB01; KBM+02].

In Section 2.2.2.2 we discussed some of the above issues from the viewpoint of devices. There we identified the general need for middleware support. Thus, complementary to Requirement 2.2.1, we state:

REQUIREMENT 2.2.10 **Support for Devices**
A middleware should offer means to provide assistance to devices. This includes computational resources as well as transparency of heterogeneity.

Finally, complementary to the requirement above we include the following requirement.

Simplicity Winograd remarked that "*the key bottleneck is the human mind*" [Win01]. Any system that requires a complex and thorough understanding of its internals by system builders in order to use its facilities is likely to fail. The time needed to achieve the necessary level of sophistication to master such a system appropriately is usually too long. Hence, like the World Wide Web demonstrates, a principle of "keep it simple" usually is more successful than other more heavyweight approaches. The HTML and HTTP protocols are simple, yet sufficient to achieve the same goals as other more powerful formating systems and communication protocols preceeding them. Thus, simplicity is a key to a successfully deployable system. Interestingly, this point of view is the very basis for the cooltown project [KB01; KBM+02]. The cooltown infrastructure relies completely on HTTP and XML/HTML as transport protocol and data format, respectively.

REQUIREMENT 2.2.11 **Simplicity**
The interface between the infrastructure and its applications should be following the rule of "keep it simple."

2.3 Summary

In this chapter we have motivated the challenges intrinsic in the new field of pervasive computing systems. We started by introducing definitions of pervasive and ubiquitous computing and highlighted important characteristics different from distributed systems. For example, different assumptions are necessary about available quality of service or that clients are attached to the network permanently. Then, we analyzed the impact of the new semantics of highly dynamic pervasive systems on the design of hardware, applications, and middleware. As a result many assumptions valid for distributed systems have to be revised and extended in order to match the requirements found in mobile environments.

Based on our own observations and a thourough review of related literature, we introduced several taxonomies, such as a classification of device mobility, the forms of possible support from the infrastructure, and the different categories of context commonly in use. These are then leveraged for the identification and formulation of a number of key requirements that must be considered when defining infrastructure support for pervasive environments. These requirements, like support for mobility, support for context-awareness, extensibility, and scalability serve as the basis for the design and assessment of the solutions developed in the remainder of this thesis.

Overall, this chapter lays important foundations, defines central concepts, and introduces central requirements, on which the findings presented in later chapters are based.

3 Publish/Subscribe Systems

The hard and stiff breaks,
the supple prevails

Tao Te Ching

3.1 Introduction

The last chapter was dedicated to the introduction of pervasive and ubiquitous computing. But, before we can dive into the technical details, we have to lay the foundations of the underlying publish/subscribe system we use as a starting point. Then, we compare this basis with the requirements identified in Chapter 2 in order to identify the necessary course of action and the potential for extensions.

Thus, the goals of this chapter are threefold: (i) it builds up the necessary foundations for understanding the publish/subscribe paradigm, (ii) it introduces the distributed notification service REBECA and most importantly, contrast its basic properties with the requirements we identified in the last chapter, and (iii) it gives an extensive review and analysis of relevant related work.

Consequently, the structure of this chapter is reflecting these goals. In Section 3.2, we start our discussion by describing the basic properties of event-driven publish/subscribe systems and the possible variations of filters and notification routing that exist in this field. The subsequent Section 3.3 introduces REBECA, the implementation of a distributed publish/subscribe notification service this thesis is based on. Section 3.4 clearly identifies the functionality missing from the basic REBECA model to be suitable for mobile, pervasive computing by comparing it to requirements defined in the previous chapter. Finally, Section 3.5 presents, discusses, and analyzes related work from the fields of event-based systems, communication paradigms and pervasive environments.

3.2 Publish/Subscribe Systems

A *publish/subscribe* system minimally consists of the following entities: producers and subscribers as interacting components, events and notifications as means of communication between producers and consumers, subscriptions as a standing request and indication of interest in certain notifications, and the event notification service as mediator between producers and consumers of notifications. The notification service thereby is responsible for guaranteeing the delivery of events to interested parties according to the subscriptions issued by subscribers (see Figure 3.1). Usually, the notification service itself is a subsystem of a larger system and specialized on message delivery and routing.

3.2.1 Events and Notifications

The term *event* is tied to a happing of interest which is observable. In general, any happening can be an event. This may be a happening in the physical world as well as something inside a completely

	Description
Event	Observation of a concrete happening
Notification	Reification and representation of an event for processing. A single event can result in a multitude of notifications
Message	Data container for transmitting notifications through the system

Table 3.1: Events, notifications, messages

virtual world of a computer system. For instance, the detection of a person in a room can be conveyed to interested listeners (a security application or an "event log"). Other examples include recurring events, like a timer event, or a change of a datum in a database triggering some actions. Events as such can model arbitrary happenings of interest on all levels of a system, from low-level hardware events ("interrupts") up to business-level events in eCommerce applications or *enterprise resource planning* (ERP) systems [Luc02].

The reification of an event in a publish/subscribe system is a *notification*. It represents the data describing the observed happening. A notification is created by the observer of the event. The content of a notification usually is application-dependent and may just indicate the plain occurrence, but it can also carry additional information describing the circumstances of the event. For example, in an active badge system, e.g., [BBHM95], events include the detected ID of the badge and the time of observation whenever a user is detected by a sensor for location tracking.

Please note that we cannot assume that an observation is reported only by a single event. Dependent on many factors the same observation might lead to multiple notifications. The notifications may carry different amounts of information supporting different views of the happening for different application domains. It may be subject to transformation or aggregation in *complex event processing* (CEP) systems, e.g., [Luc02] (cf. also Chapter 8). Or it may rely on different data models for representing event data. The most common data models are name-value pairs [CRW01], objects [BBHM95; EGD01], or semi-structured data [MF01; AF00], i.e., XML.

On the transport level described here, notifications are forwarded by *messages*, which basically are containers for data on the network level. They carry data between the endpoints of the underlying communication mechanism. The discussion above is summarized in Table 3.1.

3.2.2 Publishers and Subscribers

The clients of an event-based system act as producers and/or consumers of notifications. *Producers* emit notifications whenever an event occurs. A producer does not necessarily have to publish every single event. In general, producers are components that are self-contained. Hence, the course of action taken after an event's occurance is up to the internal computation done within the producer. For instance, in some cases events are aggregated and after a sufficient number of events are collected a new notification is generated (cf. also Chapter 8).

Whenever a notification is generated the producer "simply" publishes it into the notification service. The producer is not aware of the (potential) receipients of its notifications. This mode of *decoupling in space* is one of the major advantages of publish/subscribe systems. After publish-

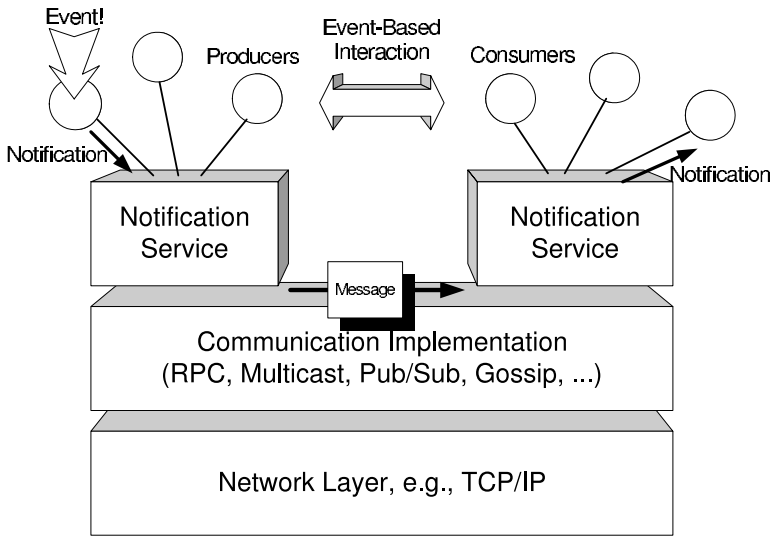


Figure 3.1: Event-based interaction

ing a notification the notification service is responsible for distributing notifications reliably to any subscriber that issued a matching subscription.

On the other end of the communication relationship we place *consumers* or *subscribers*. They issue a standing request for certain notifications. Once they receive such notifications via the notification service, they react to them, accordingly. They, too, are oblivious to the issuer of the notification. Thus, interaction is inherently data-driven. Not knowing the actual communication peer, they issue a description of the data they want to receive. This description is called a *subscription*. Different classes of subscriptions are introduced in the next subsection. It must be noted that a component can act both, as consumer and also as producer of notifications. No exclusive separation of roles is assumed. In terms of *interfaces* the subscription is the input interface to a component, i.e., a description of the data a component is able to process. Complementary, a producer may issue *advertisements*, a means to define an output interface of a component. We define advertisements in the next subsection.

Throughout the course of this thesis we assume no other communication paradigm than publish/subscribe to be in use, unless explicitly stated otherwise. A detailed discussion about the relation of publish/subscribe to other communication paradigms, such as request/reply, can be found in [FMG03].

3.2.3 Subscriptions and Filters

A *subscription* describes and represents the interest for a certain set of notifications. Consumers register their interest by submitting subscriptions to the notification service, which evaluates the subscriptions on behalf of the consumers. The intended semantics is to filter out all unwanted information and only let information pass that exactly matches a subscription. Thus, subscriptions

are commonly implemented as *filters* in the notification service. In particular, filters constitute an evaluation function that tests "incoming" notifications. Thereby, the function's range is restricted to boolean values, i.e., either a notification matches a subscription ($true$) or not ($false$).

In general, a subscription function may specify more constraints on message delivery than a pure filter function on a message's content. It also might include meta-data. This can be exploited for the specification of additional data influencing the delivery decision. For example, classes of notification can be selected based on their security credentials [BEP+03] or timing information to access past notifications [CFH+03]. In this thesis we make heavy use of this possibility and extend the notion of subscriptions and notifications to *context-dependent* subscriptions and notifications in Chapter 4.

Not necessarily part of a publish/subscribe system, but possibly helpful are *advertisements*. They complement subscriptions: where subscriptions state an interest for certain information, advertisements announce the kind of information a producer is able to produce. This complementary nature usually is exploited to optimize message routing in the infrastructure because the notification service "knows" what data can be expected from which producer and set up the network appropriately.

3.2.3.1 Filter Models

Obviously, the expressiveness of a subscription is dependent on the specification language used. In distributed notification services, essentially five *filter models* are distinguished: channels, subjects, types, content-based, and concept-based.

Channels. *Channels* are the simplest form of subscribing to sets of notifications. In the channel-based model, a producer has to select a named channel into which a notification is then published. For selecting certain notifications the client wants to receive, it only can select a channel. Any information published on this channel is delivered to the client; independent of the *concrete* interest of the client. An example of this approach is the CORBA Event Service [Obj00]; the CORBA Notification Service [Obj02] also relies on channels but additionally offers filters on a notification's content.

Subject-based addressing. *Subject-based addressing* uses string matching for notification selection [OPSS93]. Every notification is part of a hierarchy of subjects. I.e., every notification is annotated with a *character string*, describing the position relative to the hierarchy this data item belongs to. For example, an application might publish some data related to *Honey* under the subject "`/The Hundred Acre Woods/Winnie the Pooh/Stock/Honey`". However, another (correct) view might be constituted by publishing the same data under the branch:
"`/The Hundred Acre Woods/Stock/Winnie the Pooh/Honey`".

Each level "down" in the hierarchy describes a finer granularity of notifications and thereby a smaller subset of all notifications in the system. The closer to the root node, the more general the selection criteria gets.

Type-based selection. *Type-based selection* uses similar path expressions and sub-type inclusion tests to select notifications [BBMS98; EGD01]. With multiple inheritance, the subject tree is extended to type lattices that allows for different rooted paths to the same node. Often, type checking is complemented with content-based filters to improve selectivity.

Content-based filtering. *Content-based* filtering is the most general scheme of notification selection [CRW99; Müh01]. Where other approaches use distinct addressing schemes for notification

	Description
`publish`(N)	Publishes event observations into the event system.
`subscribe`(Sub)	Subscribes to certain information.
`unsubscribe`(Sub)	Unsubscribes to certain information.
`notify`(N)	Notifies a client about the arrival of a notification N matching a previously issued subscription Sub

Table 3.2: The publish/subscribe interface of a event notification service

selection (e.g., *strings* for subject specification), content-based addressing uses the complete content of a message as possible selection criteria. Boolean expressions evaluate the whole content of notifications, where the data model of the notifications and the expressiveness of the predicates determine the filter selectivity. Available solutions range from template matching [CDF01], simple comparisons [CRW01] or extensible filter expressions [MF01] on name-value pairs, to XPath expressions on XML [AF00] and arbitrary programs and mobile code [DMDP03].

Concept-based filtering. *Concept-based* filtering [Cil02; CBB03] is another general scheme of notification selection and an extension to content-based filtering. It is especially useful in environments with heterogeneous data sources where the semantics of data in the system is not defined clearly. There, filtering has to be done on a level where sematic translations have to be performed in order to identify matching filter/notification pairs. Semantic translations usually employ meta-data and are based on ontologies. Hence, concept-based filtering introduces much flexibility on the one hand, but limits its applicability to domains where well-defined ontologies exist.

3.2.4 Event Notification Service

Because publish/subscribe is intended to decouple producers and consumers of information a mediator between the participants is needed. An event notification service, or notification service for short, can implement this role. In event-based systems, as we explore them in this thesis, the notification service alone is responsible for message delivery from publishers to subscribers. We have shown this in Fig. 3.1.

The notification service offers a simple, yet sufficient, publish/subscribe interface for clients. Only the *publish*, *subscribe*, *unsubscribe*, and *notify* calls are needed (cf. Table 3.2). Messages get into the notification service by a *publish* call of an attached client and publisher [1]. The notification service then tests the newly arrived notification against all subscriptions which are currently active in the system[2]. Active subscriptions are issued by some consumers, stating their interest by issuing a standing request, using the *subscribe* call. The notification service then adds a new subscription to the set of active subscriptions. Whenever the test of a notification against an active subscription is

[1] Please note that a notification service is not necessarily a single process running on a single CPU. The notification service assumed in this thesis is a distributed notification service and consists of a network of notification brokers. Cf. next Section.

[2] Again, this can mean that this step is done in a rather sophisticated and distributed fashion within a broker network.

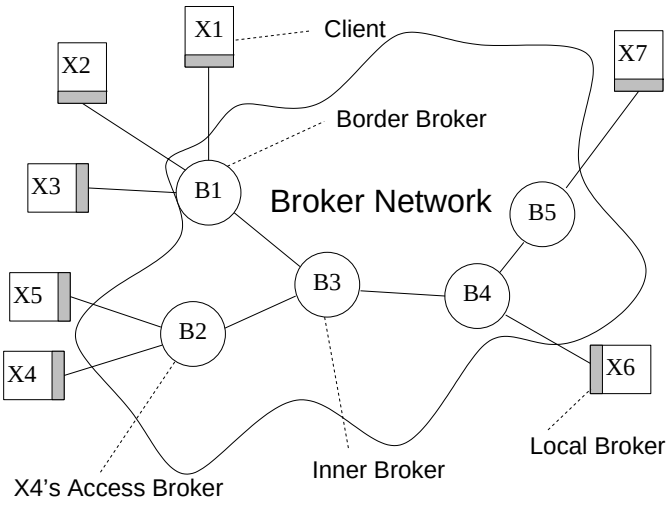


Figure 3.2: The router network of REBECA.

positive, the notification eventually is delivered to a subscriber. Delivery is done by calling the *notify* call of a registered client.

Obviously, from the application's point of view the mode of operation of a notification service is opaque. Whether the functionality is centralized or distributed does not matter to an actual client. The only way to access the classical model of a notification service is by its interface, thereby effectively constituting the "black-box model" as introduced in Section 2.2.4.2.

3.3 The REBECA Model

This section describes the system model and the basic characteristics of the REBECA notification service [FM00]. It implements the publish/subscribe interface as described in Section 3.2.4 above.

In Figure 3.2, we have depicted the underlying system layout of REBECA. Basically, the architecture is centered around a distributed *network* of communicating *notification brokers*. Because of its distributed nature, REBECA is a representative example of a distributed notification service like SIENA, JEDI, etc. REBECA supports different routing algorithms and data and filter models. A detailed description of the general impact on message delivery compared to other notification services is beyond the scope of this thesis and we refer to [Müh02; MFGB02] for a detailed description.

In this thesis REBECA is used as the underlying system model and testbed for the extensions proposed in this thesis. However, we tried to specify any extension to the notification service in a way that specific characteristics of REBECA are not necessarily part of the proposed approach in order to foster versatility.

3.3.1 System Model

For REBECA a concurrent program execution model of communicating processes is assumed [LL90]. Processes interact by message-passing over links between the processes concerned. Any link connects a pair of processes. Message passing is asynchronously as defined, e.g., by Mattern [Mat89]. Hence, there is a certain time span between the $send()$ and $receive()$ operations on the link. Links are assumed to exhibit no failures and to obey FIFO (first-in-first-out) ordering of messages. Consequently, no messages are lost or corrupted due to link failures and messages are received in the same order they were sent. The prime advantage of a nomadic computing system (Section 2.2.1) here is that it is reasonable to assume that such guarantees are maintainable in the infrastructure. For other settings initial solutions are available and might be applicable [CMPC03; IGE+03; Müh02]. However, the above mentioned properties are hard to maintain at the borders of the system where clients are mobile. Solutions for maintaining FIFO ordering and avoidance of message loss in the face of client mobility are presented in Chapter 5.

The underlying system model is build around a separate logical network of communicating processes on top of a concrete network topology. We assume a set of processes located at some physical node in a network and connected by point-to-point connections realized by the underlying network technology. For instance, common internet technology easily meets those requirements. A set of machines connected by the Internet host a set of processes which in turn are connected by TCP/IP conncetions between the host machines. But, to facilitate extensibility and flexibility, the system model is separated from the actual implementation. Therefore, other connection technologies are viable choices, too. For instance a multicast-based solution might be used to improve communication performance.

3.3.2 Architecture

Effectively, the role of the REBECA notification service is to decouple sender and recipient of notification messages. This is done in a—for clients—transparent way. The perception of a client is that of a single point of communication, i.e., the actual implementation of the publish/subscribe interface and the internal architecture of the service are opaque. In particular, whether the notification service is centralized or distributed is of no concern to the clients. Thus, the REBECA architecture of a distributed network of interacting event brokers as shown in Figure 3.2 can be optimized in various ways. We will exploit this particular advantage of a distributed architecture throughout the next chapters of this thesis.

The communication topology of the overlay network of event brokers is described by an acyclic graph (Fig. 3.2). Edges are point-to-point connections in the underlying network, for instance, TCP/IP connections. The acyclic graph used is comparable to the single spanning tree approach of multicast algorithms [DC90]. We are aware that a single tree is a potential bottleneck and bears the potential to "act" as a *single point of failure*. Nevertheless, as we will show in later chapters, this model can be of great use in the context of this thesis. Therefore, we believe that adhering to the model of a single spanning tree is beneficial. Moreover, promising approaches for tackling problems of scalability and redundancy are under way [CMPC03; TBF+03; PB02].

Please note that the original architecture of REBECA was designed for scalability and notifiaction routing optimizations. In this thesis, we draw from such built-in capabilities where possible and the underlying architecture is not changed. Our approach rather is to add extensions to this basic model for proper support of mobile and pervasive applications and leave the basic functionality and properties untouched where possible.

For the structure of the broker network, besides the characteristic of being an overlay network, three types of brokers can be distinguished: local, border, and inner brokers.

- *Local broker.* Local brokers act as access points to the infrastructure[3]. Typically, they are part of an application's communication library and are loaded on application startup. Thus, they cannot be handled as regular part of the broker network and they do not show in the actual graph structure of the notification service. A local broker is connected to a single border broker.
- *Border broker.* Border brokers are always the first "hop" into the network of brokers and form the boundary of the routing network. In this thesis, border brokers play a major role for supporting and hosting mobile clients, as well as maintaining caches and connections to their local brokers.
- *Inner broker.* Inner brokers are connected to other inner or border brokers and do not maintain any connections to clients.

All brokers support *content-based filtering* as introduced above in Section 3.2.3. In the distributed case content-based filtering is also exploited as routing decision within the network. This is detailed in the following section.

3.3.3 Content-Based Routing

Contrary to centralized systems, in distributed notification services notifications do not only have to be matched against a single table of subscriptions but also are routed through the network to appropriate local brokers for delivery to applications. In a centralized implementation matching of notifications is reduced to efficient matching algorithms [YGM94; MFB02]. Although sufficient at first sight, in large scale scenarios—and we consider pervasive computing environments as being large scale (cf. Sect. 2.2.4)—a centralized solution poses the risk of being a bottleneck because of the possibly intense consumption of bandwidth and computational resources. A distributed solution therefore is more flexible and desirable. In such systems, the same operation, i.e., matching of notifications and subscriptions must be done in every broker on the delivery path between producers and consumers of data. On the other hand, this solution naturally distributes the tables of subscriptions throughout the network and, in some relevant scanarios considered in this thesis, massively can draw from message delivery localities if producer and consumer are "close-by" according to the metrics of the network (cf. Chapter 7). Moreover, a result of our discussion of pervasive computing environments in Section 2.2.4 is that interactions take place within a local *boundary* as well as across boundaries[4]. Hence, within a boundary interaction can be realized using a centralized approach, but across boundaries it is distributed.

The matching decision then is used to identify the appropriate next hop in the network towards the consumer. Filter-based routing depends on routing tables that are maintained in the brokers and contain *filter-link* pairs indicating in which direction matching notifications have to be forwarded. Entries in the tables are updated by sending new and canceled subscriptions through the broker network, adding or deleting (F, L)-pairs that contain the filter and the link from which it was received. Every incoming notification is tested against the routing table entries to determine the set of links with matching filters, omitting the link from which it was received. In a second step the notification

[3] Please note that they are sometimes called *access broker*, as well.
[4] Cf. the *boundary principle* on page 26.

is forwarded to the respective neighbor brokers. If the incoming notifications of each link are routed sequentially an end-to-end sender FIFO characteristic holds.

In the following we introduce the different routing strategies built into the REBECA notification service. However, we consider the details of each strategy to be out of scope of this thesis and instead refer to the appropriate literature, here especially to [Müh02]. To keep the solutions presented in our work as versatile as possible, we assume *simple routing* as the most general routing strategy to be used. Where appropriate, in later chapters, we discuss the impact on other routing strategies qualitatively.

Different flavors of filter-based routing exist, which differ in their strategy to update the routing tables.

- *Simple routing.* Simple routing assumes that each broker has global knowledge about all active subscriptions. It minimizes the amount of notification traffic, but the routing table sizes in each broker can grow significantly. Every (un-)subscription has to be processed by every broker resulting in a comparable high filter forwarding overhead. In large-scale systems more advanced routing algorithms may be applied to exploit commonalities among subscriptions in order to reduce routing table sizes [MFGB02]. REBECA includes three of them [Müh02], identity-based routing, covering-based routing [CRW01], and merging-based routing [MFB02].
- *Identity routing.* Identity-based routing avoids forwarding of subscriptions that match identical sets of notifications.
- *Covering-based routing.* Covering-based routing avoids forwarding of those subscriptions that only accept a subset of notifications matched by a formerly forwarded subscription. It might be necessary to forward covered subscriptions along with an unsubscription if a subscription is canceled.
- *Merging-based routing.* Merging-based routing extends covering further. Each broker can merge existing routing entries to a more general subscription, i.e., the broker creates a new cover for the merged routing entries replacing the old ones. Only the resulting merged filter has to be forwarded to neighbor brokers, where it covers and replaces the existent original filters. Merging can be either *perfect* or *imperfect*. Perfectly merged filters only accept notifications that are accepted by at least one of its original filters. On the other hand, imperfectly merged filters may accept more notifications than their original filters separately would. Imperfectly merged routing table entries may lead to increased network traffic but allow for *lazy updates*, hiding frequent reconfigurations in covered parts of the network.

Advertisements are an additional mechanism to optimize subscription forwarding. Subscriptions are only forwarded into those subnets of the overlay network where a producer has issued an overlapping advertisement. Whenever a new advertisement is discovered, overlapping subscriptions are forwarded appropriately. Similarly, if an advertisement is revoked, it is forwarded, and subscriptions that can no longer be serviced are dropped. Advertisements can be combined with all routing algorithms discussed above.

3.4 Missing Functionality

The preceding sections have introduced the basis on which we want to build extensions to facilitate mobile and context-aware applications. In this section, we therefore focus on the shortcoming of

the current model and architecture of event notification services in the light of the requirements we definined in Section 2.2.4.

Due to the inherent properties of notification services, some requirements are already met, some are at least partly covered. For example, loose coupling is one of the most basic characteristics of publish/subscribe. Hence, the choice of relying on the data-driven, asynchronous publish/subscribe paradigm is already justified. But, on the other hand, not all characteristics of pervasive environments have been given thought to. In particular, the basic architecture described above does not have built-in support for mobility and location-awareness. The prime requirements we have identified in Section 2.2.4!

Therefore, in the remainder of this chapter we discuss the basic issues involved when adding mobility support and location-awareness to a publish/subscribe infrastructure. Then, we sketch the impact on the architecture of a publish/subscribe infrastructure supporting both. Later in this thesis, this leads to Chapter 5, detailing the necessary extensions to the basic publish/subscribe model for "classical" mobility support. Consequently, Chapter 6 and Chapter 7 detail necessary extensions for location-awareness in the infrastructure, besides other things.

3.4.1 Mobility

In Section 2.2.4 in Requirement 2.2.5 on Page 23, we identified the need for mobility support to make a publish/subscribe infrastructure usable in mobile and pervasive settings. There, we stated the need for two orthogonal forms of mobility support: transparency and awareness of mobility at the same time. Some scenarios require the former, some the latter and some both types of support. For instance a so-called legacy application, like a stock quote monitoring application, is not aware of the mobile client it is running on. Moreover, the notifications received usually have no relation to the current location. Like other solutions for network transparency, e.g., mobile IP [Joh95], the actual details are hidden in the network layer. Here, the network layer is the notification service. Thus, we need to provide an extension for the basic publish/subscribe notification service that facilitates *transparency* for the client. The same mechanism comes into play whenever a mobility-aware application wants to delegate mobility aspects of its operation into the infrastructure. This may be the case for forms of context-sensitivity that relies on aspects other than the current location. For example, notifications should only be delivered in a certain time frame, there the context is *time*, independent from the current location of a client.

Obviously, a naïve solution for transparency is to rely on a sequence of *subscribe-unsubscribe-subscribe* operations for the same information. As we show later in this thesis this unavoidably leads to unwanted message loss. Therefore, in Chapter 5 we introduce an extension to the standard REBECA notification service that handles relocations of clients transparently without losing the guarantees REBECA offers for the non-mobile use case.

Mobility-awareness on the other hand is a necessary option for every context-sensitive application. In order to meet Requirement 2.2.7, i.e., the support and fostering of adaptive behavior, we have to ensure that an interested client is able to access information about its current whereabout. Most contemporary implementations of the publish/subscribe paradigm do not support mobility-awareness. Awareness is a feature of the applications. On an application domain-specific level, i.e., when consumer and producers share the same common knowledge about their environment (and are able to provide this information at all), it is possible to process such information on the shared common ground. Information about location then is encoded in the actual content of a message. The drawback of such solution is obvious: a roaming client always is forced to update its own position by

issuing a new subscription with its new position, together with an unsubscription at its old position. Besides possibly violating our requirement of simplicity, such position handling has a significant impact on the computational resources of a small device. Moreover, the routing network needs time to propagate information about a new position through the network. This leads to a perceivable slow down of message delivery. In a fine grained location model, for instance a floorplan, where location updates occur often, this behavior significantly impedes the usability of an application.

More desirable is to put mechanisms in place where the client can specify a *location marker*, acting as a placeholder for the client's current position. Whenever the location is updated, either by a client or an external tracking- or location service, the subscription of this client is adapted automatically. Without going into details, this imposes some interesting challenges which are addressed in Chapter 6 and partly in Chapter 7. To implement mechanisms for automated location-awareness, we have to separate the *location model* in use from the actual application domain it is used in. For efficient routing and matching of filters carrying a location specification, we need to make *location* a first-class concept of the routing network. To avoid issues of heterogenous location models and semantic transformations, as a common basis, we introduce a *reference location model* for the use in the infrastructure in Chapter 4.

3.4.2 Dynamics, Adaptivity, and Reactive Behavior

Our analysis of requirements in Chapter 2 identified a number of requirements addressing the need for a flexible and extensible infrastructure in order to handle the inherent dynamics of pervasive systems. By design, a notification service and the data-driven publish/subscribe paradigm are well suited to meet the requirements stated. In Chapter 8 we discuss this in the light of the development of context-sensitive applications. However, some extensions to the basic model presented in this chapter are needed to properly address the challenges (see also Section 3.4.5 and Section 3.4.6 below).

3.4.3 Scalability

The choice of a distributed notification service as message routing infrastructure, to some extent, is motivated by its potential to host a large number of clients, i.e., its inherent scalability. One of the characteristics of a pervasive environment is its scale in space as well as in the number of producers and consumers of information (Requirement 2.2.9 on Page 26). Therefore, a distributed notification service already accomodates scalability. But the mechanisms presented in this thesis, here in particular the algorithms we discuss in Chapter 5 and Chapter 7, have been designed to draw from message delivery localities in the broker network, promoting scalability even further.

3.4.4 Decoupling in space and time

Another important issue and the main focus of Chapter 7 is to provide means to meet Requirement 2.2.8: decoupling of clients of the publish/subscribe notification service in space and time. The publish/subscribe paradigm, by design, already decouples sender and receipient of data in space and—to a certain extent—in time. Decoupling in time is only possible for the time it takes for the network to propagate a notification through the network. Then a message is "buffered" in the network. After this, the notification is gone if not special facilities for buffering notifications are put in place to provide for longer lasting persistence. In the general case, "past" notifications are not accessible for clients that, for instance, were switched-on "just a fraction of a second" too late to

observe an important event. Unfortunately, in a nomadic system setting this is a common situation. Clients are assumed to roam freely and to frequently connect to different new environments. Any such time the scenario above can happen. This might be intolerable for clients, which have to rely on a series of notifications for adaptation to a new environment or simply for bootstrapping and reaching a consistent state of operation. As the commnication is asynchronous and event-driven, a client cannot assume that the needed information is (re-)published in due time. Here, special buffers, providing at least for a speed-up of the bootstrapping process can be of great help. Chapter 7 details our solution to the problem stated above.

3.4.5 Application support

The concern of Chapter 8 is the definition of a conceptual framework with which it is possible to define *control-driven* applications in the face of the purely *data-driven* publish/subscribe paradigm (e.g., [PA98; PLM02]). Often, finite state machines (FSM) are used to specify behavior on a high semantic level. Such FSMs are then used to model state changes and the reaction to *events* happening outside the application. In Chapter 8 we show how this approach can be applied to the development of context-aware applications and how we can exploit the characteristics of the notification service to support this approach. Thereby, we show how the publish/subscribe paradigm can be exploited and extended to provide reactive behavior (Requirement 2.2.2).

3.4.6 Support for small devices and simplicity

By allowing the specification of filters, as introduced in Section 3.2.3, together with the easy-to-use interface to the notification service, we already meet the Requirements 2.2.10 (support for small devices) and 2.2.11 (simplicity) to a large extent. Filters significantly can reduce unwanted traffic a client otherwise might receive. The simple-to-use interface makes it easy to integrate heterogenous devices into environments using the publish/subscribe paradigm.

In addition, throughout the remainder of this thesis we make sure that every extension made to the notification service is in accordance with the requirements above. Especially in Chapter 6, we propose an extension to the basic notification service that tries to minimize the number of unnecessary notifications sent to a client compared to network flooding; the only alternative solution guaranteeing the same semantics. The proposed solution thereby avoids the danger of network congestions on a low-bandwidth wireless connection and overloading the computational resources of a resource-limited device. Moreover, in Chapter 8, we allow clients to place *aggregators* and *interpretors* (see Section 8.3.3) in the network, thereby relocating a significant amount of traffic and computation into the infrastructure.

3.5 Related work

In this section we introduce and analyze work related to the work presented in this thesis. We structure the discussion that follows along two main topics: first, we introduce common related work from the field of publish/subscribe notification services and coordination models. Then, related work from the field of pervasive computing is analyzed. We restrict our discussion to work with a focus on middleware and event-based communication.

3.5.1 Notification Services

A considerable amount of notification services exist and many concrete systems have been designed and implemented, both in industry and academia. In this subsection we give a representative overview over existing solutions for asynchronous, event-driven communication.

3.5.1.1 CORBA Notification Service

The CORBA Notification Service is an API specification as part of the Common Object Services of the CORBA plattform [Obj02] and supersedes the previous event service specification [Obj00]. It relies on channel-based addressing (Section 3.2.3.1), i.e., publishers have to get a reference to a specific channel and send their event data as notifications into this channel. Consumers, on the other hand, select a channel and additionally specify a content-based filter for client-side filtering on data published on this channel. Visibility of data is restricted to the channel it is sent on. An additional problem inherited from the channel-based approach is the structure of channel names and, in particular, the relationship between channels. Part of the semantics of the data sent using a channel already is "encoded" in the channel's name. This introduces the problem of "channel discovery," which seriously restricts its general usability in dynamic and mobile scenarios, where heterogenity has to be considered. Producers and consumers of notification data have to deal with channels explicitly, thereby adding to the complexity of applications. Another issue is the scalability of a centralized notification service.

However, for environments where one can make assumptions about the clients of the infrastructure and the applications needed, the CORBA notification service is a popular choice. We will discuss one such system examplarily in the context of GAIA (Section 3.5.4.2 on Page 49).

3.5.1.2 Java Message Service

The Java Message Service (JMS) is an API specification being part of the Java2 Enterprise Edition (J2EE) [Sun02a; Sun03]. Different to the CORBA solution, JMS can be used without the enterprise object platform it is a part of. The JMS uses so called *topic-based subscriptions*, which stands for message grouping according to abstract topics *plus* content-based filtering on a set of header fields and properties, similar to CORBA channels. Thereby, JMS provides an API layer that can be put on top of many industry messaging and publish/subscribe products, including the Corba Notification Service. The semantics of topics is left to be defined by the actual topic provider, called JMS provider. The same applies to the topic's syntax and management. Topics can be grouped in various ways, the most prominent hierachy, probably, is based on a Java packages-like structure, i.e., a dotted notation: `winnie_pooh.stock.honey`. JMS providers can choose for what position in a subject hierarchy their notifications are for. This can be a leafe node, as well as a complete subject subtree.

As JMS basically is an API only, it is extremely versatile and flexible. Therefore, JMS is probably becoming the most popular messaging API. Many projects, including industrial, academic, and open source, are implementing the JMS API. A partial implementation of the publish/subscribe part of JMS is part of the REBECA notification service, thereby, combining the potential of JMS with the advantages of distributed notification routing.

3.5.1.3 Gryphon

The Gryphon project's goal at IBM Research [IBM01a] was the development of an industrial-strength, reliable, content-based event broker. Later Gryphon was integrated into IBM's WebSphere suite as the IBM WebSphere MQ Event Broker [Cor03]. This indicates that Gryphon is a mature and successfully deployed event service. It provides a redundant, topic- and content-based multi-broker publish/subscribe service. It includes an efficient event matching engine, a scalable routing algorithm, and security features. Naturally, mobility and context-awareness are not addressed explicitly.

Gryphon is based on an information flow model for messaging [BKS+99]. The exchange of information between producers and consumers is specified in a so called *information flow graph* (IFG). Information flows can be influenced by (a) filtering, (b) stateless transformations, and (c) stateful transformations, i.e., aggregation. Therefore, Gryphon may be a viable choice for the model proposed in Chapter 8.

Three types of brokers are distinguished: *publisher-hosting broker* (PHB), *consumer-hosting broker* (CHB), and *intermediate broker* (IB). A PHB hosts a number of publishers entering data into the IFG. Complementary, a CHB hosts consumers of data, and an intermediate broker has neither publishers, nor consumers attached. A drawback, however, can be seen in the static deployment of the broker network.

Interesting in the context of Chapter 7, i.e., decoupling in time, is that a recent addition to Gryphon added a durable subscriptions service [BZA03]. It added exactly once delivery semantics over periods of disconnectedness of a client. This means that the event stream is buffered and replayed upon reconnection. Therefore, an event log is held at the PHB.

3.5.1.4 Research Prototypes

In this subsection we present a selection of research prototypes in the field of distributed publish/subscribe notification services we consider to be relevant for the work presented in this thesis.

CEA. The *Cambridge Event Architecture* (CEA) [BBHM95; BMB+00] originated in the early 90s. Its roots lie in the need to address asynchronous communication for sensor-rich applications. In the beginning CEA was tightly integrated in the standard middleware CORBA. Middleware clients acting as event producers or consumers were standard CORBA objects. An event producer registeres the kind of events it produces with a name service, thereby effectively using an *advertising* mechanism.

In the standard *synchronous* middleware approach of CORBA, an event consumer directly registers with the producer, after doing a *lookup* for the kind of notifications it is interested in. The producer in turn, notifies all registered clients whenever an event occurs. Obviously, direct communication introduces tight coupling, which is not desirable in general. To address this issue, CEA introduced intermediaries, so called *event mediators*. They implement both, the producer and consumer interfaces, acting as buffer. The data model does not include general content-based publish/subscribe. The design goal of CEA was to integrate with standard middleware. Therefore, events are strongly typed objects. Initially, subscriptions were template-based for equality matching only. This was extended later to facilitate a more general predicate-based language with key/value pairs. Furthermore, CEA supports composite event patterns [HBBM96].

Although event mediators can be federated or chained, they lack the content-based routing mechanisms of other systems. This seriously limits the scalability of the system as subscribers are forced to know the producers of a certain event type. Particularly in mobile settings this can be seen as a major drawback.

HERMES. HERMES [PB02; Pie04] is a very recent implementation of a distributed content-based publish/subscribe system developed at the Computer Laboratory at the University of Cambridge. HERMES follows a type- and attribute-based publish/subscribe model including type-checking of events and event-type inheritance. For the use in more dynamic environments, HERMES uses a *peer-to-peer* overlay network on top of a regular network infrastructure. This is comparable to the concepts we started to explore in [TBF+03]. Routing algorithms are introduced that are implemeted on top of a *distributed hash table*; the common data structure for peer-to-peer systems. The goal is to achieve reduced routing state in the system and to achieve a certain degree of fault-tolerance with the prospect of ad-hoc networking.

This core functionality of the distributed event notification service is augmented by three higher-level services that address different requirements: congestion-control and recovery to foster scalability and fault-tolerance, composite event detection for the detection of complex events, and security.

SIENA. SIENA [CRW01; CRW00] is a publish/subscribe event notification service. As REBECA, it is implemented as distributed network of servers to achieve scalability for a large number of communicants and high volumes of notifications spread across a wide-area network. The underlying system model is almost identical to the one we presented in Section 3.3.1 on Page 35. SIENA servers act as access point (in REBECA called border brokers) and as store-and-forward network routers. The access point routers offer clients the usual publish/subscribe interface to the distributed event service. Additionally, publishers can use the access points for *advertising*, i.e., the publication of information about the notification they are going to publish afterwards. Complementary, subscribers use *filters* to express their interest in a certain kind of information.

The underlying *notification data model* is based on *typed filters*, i.e., a notification in the model is a set of typed attributes. Each individual attribute consists of a triple $(type, name, value)$, describing the content the filter shall match. Noteworthy is that the $type$ attribute belongs to a *predefined* set of types available in the system.

The basic SIENA model, like the basic REBECA model, offers no explicit support for mobile clients and context-aware applications. The mobility extensions of SIENA in [CIP02] are very similar to the JEDI approach we describe below. Explicit sign-offs are required and stored notifications are directly requested from the old location, too, resulting in possible message-loss or duplicates. We discuss the drawbacks of such solutions in full detail in Section 5.5.2 on Page 92, in the light of our own proposed algorithm.

In a recent technical report [CCW03] the authors describe a mobility extension to SIENA, based on a general purpose support service for mobile applications as an orthogonal service to the event service. The underlying idea is to use a client proxy that is designed "to reduce the loss or duplication of information during the switch-over period."

To the best of our knowledge, support for location-aware or context-aware applications is not built into the routing infrastructure.

JEDI. The *Java Event-Based Distributed Infrastructure* (JEDI) [CDF01] is a Java-based implementation of a distributed content-based publish/subscribe system from the Politecnico di Milano. JEDI is build on the notion of active objects, acting as publishers and subscribers that exchange messages through an event dispatcher, which routes events. The event dispatcher is a logically centralized component that can be implemented by a distributed set of dispatching servers connected in an acyclic topology, i.e., a dispatcher tree. Routing is performed according to a hierarchical subscription strategy. Subscription propagate upwards in the tree and state about them is maintained in and

by the event dispatchers. Events are basically handled identically to subscriptions but are also propagated "down" along a branch with a matching subscription. Advertisements, to restrict notification forwarding, are not part of the JEDI model. There is no single event dissemination tree for all subscriptions but instead a *core-based tree* [BFC93; Bal97] is constructed dynamically. Therefore, it is necessary to determine a group leader, called the core, which announces its existence via broadcast. This introduces a rather complex protocol for event dispatchers to become part of a certain group. Each event dispatcher has to have global knowledge about all available group leaders at all times. On a request of an event dispatcher to become part of a certain group the group leader delegates the request downwards to an appropriate dispatcher, that becomes parent of the new node in the tree.

JEDI offers support for mobility [CNP00; CD01] through the notion of a `moveOut` and `moveIn` operation. The basic idea is to have a client to detach from the infrastructure by explicitly sending a `moveOut` message, serializing its state, and reconnecting to a (possibly) different broker, where a `moveIn` message is sent. Messages received at the old broker's location are requested directly by the new broker the client is attached to. This, again, can lead to message loss and duplicates. Hence, certain delivery guarentees can not be maintained. Additionally, it is not detailed what the effect of movement on the publish/subscribe infrastructure is. Moreover, as with SIENA, the proposed mechanism leads to unwanted message loss and/or duplicates. Finally, in mobile, wireless settings the notion of an explicit `moveOut` operation seems unrealistic in the face of sudden connection loss or spontaneous switch-off of a device for some reason. Also, only mobility transparency is supported, but not awareness.

Elvin. Elvin [SA97] is a notification service developed by the Distributed Systems Technology Center in Australia. Originally, the intended use was focused on distributed systems monitoring. Additions include a security framework, internationalisation, and pluggable transport protocols [5]. Content-based routing of events [SAB+00] has been added as well. Events are attribute/value pairs with a predicate-based subscription language. A publisher can request information about consumers of their published data. This feature can be exploited to stop notification production in the case no consumer is attached to the event service. This reduces computation and unnessecary communication. More recent work includes investigation about the usability of Elvin in more dynamic settings [SAS01].

3.5.2 Tuple Space based Middleware

The characteristics of mobile wireless systems favor decoupled operation and an opportunistic style of communication. Synchronous communication as it is supported by many traditional middleware systems must be replaced by asynchronous communication, e.g., to facilitate for disconnected operation. One possible alternative solution to the event-based paradigm is the use of *tuple spaces*. Although, initially not built for the use in mobile settings—the origin goes back to Linda [Gel85; ACG86]—but as a communication language for concurrent programming, tuple spaces inherently have many useful facilities for the use in mobile settings. However, as it is true for event systems, traditional tuple spaces also have to go a long way to be really adapted to meet mobile systems' requirements.

For instance, in Linda a tuple space is globally shared by all communicating processes. Memory space is addressed associatively and acts as a repository of data structures for interprocess communi-

[5] Although not further detailed above, REBECA implements so called `EventTransports`. They are meant as a plug-in mechanism for various transport protocols.

cation. The only data abstraction known are data-centric tuples. They are created and/or consumed by processes. Tuples are anonymous, i.e., independent from the process that generated them. Their selection is solely based on the data they represent. Only three different operations are originally provided by a tuple space: `write`, `read`, and `take`[6]. The `write` primitive is used to add tuples to a tuple space, `read` and `take` are used to read a *copy* of a tuple or to consume the original tuple from the tuple space, respectively. While the former leaves the original in the tuple space for other processes to read, the latter deletes the tuple from the space. Both operations are blocking, i.e., the process waits for the generation of a tuple that matches the pattern provided by the `read` or `take` operation. In principle, this is enough to provide decoupling in space and time. Producer and consumer of data do not necessarily have to be available at the same time. However, in mobile settings many problems remain: how can the model of a globally shared data space be scaled to large settings where the data space has to be distributed itself? How is consistency maintained? How can mobile hosts access a tuple space without knowledge about the environment?

In the following we want to review a representative selection of tuple spaces that have been devised also for mobile settings.

Lime. In Lime [MPR01], the shift from a fixed context to to a changing one is done via the distribution of tuple space functionality to a number of separate tuple spaces. Each tuple space is *permanently* and exclusively associated with a *mobile unit* and is called *interface tuple space* (ITS). The term mobile unit can refer either to a mobile agent, or to a mobile device. In the first case the unit is mobile logically, in the second case it is mobile physically. However, in either case, the interface tuple space is transferred with the code or device, thereby introducing the necessity to support code mobility.

A mobile unit accesses a system using conventional tuple space primitives. The fundamental idea is that each device uses the ITS as means to share data. Whenever mobile units meet, the ITS connect to each other and every tuple contained in one of the ITSs is accessible by all other ITSs and hence mobile units. Thereby, a *transiently shared tuple space* is generated. Upon arrival or departure of a mobile unit the virtual "global" view of tuples is recomputed, called *engagement* and *disengagement*, respectively, of tuple spaces.

Overall, mobility, in the sense as Lime defines it, is reduced to a view of data shared in the "global" transient tuple space. This view may be beneficial for the development of applications but it may also be too restrictive in domains where higher degrees of context-awareness are needed.

IBM TSpaces. TSpaces [WMLF98] is the IBM version of a tuple space. The design of TSpaces reflects the goal to make TSpaces usable on a wide variety of platforms. It distinguishes three important roles: client, server, and tuple space. The client basically is a rather simple class library loadable into a client application on a device and, like the whole system, is written in Java. The server is a configurable container, providing the environment for maintaining a set of tuple spaces. This is different to other tuple spaces where a single tuple space is used. In a certain sense, tuple spaces in TSpaces resemble channels or topics as known in event-based systems. The content and semantics of each tuple space is meant to be conveyed by the name of a tuple space. Information about all available tuple spaces is kept as tuples in a special tuple space called *Galaxy*.

A client which wants access to a certain tuple space must have the address of the server, as well as the name of the tuple space in order to write to or read/take from a tuple space. This can be considered a drawback of this model. However, on the server-side, TSpaces offer rather flexible and extensible

[6] Please note that there does not exist a notification mechanism.

mechanisms. From simple unstructured tuples, automated generation of "XML tuples" from XML formatted documents, to transactional behavior the *server* can be configured to support a wide variety of tuples. Moreover, like most contemporary tuple space solutions, TSpaces supports asynchronous notifications and offers subscription mechanisms for non-blocking read/takes. However, the server functionality is centralized and requires a larger amount of computational resources. Federation and distribution of tuple spaces to several servers is not explicitly supported. Additionally, sudden disconnection of clients is considered a permanent fault and results in removal of the client, thereby limiting the usability in mobile settings without "third-party" support for catering to the mobility issues.

JavaSpaces. JavaSpaces [Sun02b] is part of the Jini framework since its first introduction in 1999 [Sun99a]. Jini originally was targeted at small and mobile devices but ultimately failed to reach this goal, due to the need to have a complete Java2 runtime environment on a device. However, some efforts were made to circumvent this requirement [Sun01; ADH+99; AKZ99]. JavaSpaces itself is built as a service in a Jini federation. It uses the underlying Jini framework for various functions: service advertising [Sun99b], remote eventing [Sun99a], distributed transactions [Sun99c], and leasing [Sun99a]. For a complete overview of Jini we refer to [Edw99].

In comparison to TSpaces, the most striking difference is that only a single tuple space is used. But, on the other hand, in the Jini model of federated services, a set of different instances of JavaSpaces can live in a single environment as separated services. Thereby, different views and specialisations can be achieved. However, a client then is responsible by itself to find, select, and publish information in all JavaSpaces it needs.

The operations offered are comparable to those offered by TSpaces. The usual `write`, `read`, and `take` operations are provided. Additional non-blocking variants are available, employing the underlying Jini Remote Events mechanism for peer to peer eventing based on Java RMI. Obviously, using synchronous, tighly coupled communication for eventing is limiting the usability in mobile settings seriously. Remarkable is the leasing mechanism. JavaSpaces uses time based garbage collection on tuples in the tuple space. A `write` operation always returns a so called *lease*. A lease is a token describing the time a tuple is guaranteed to be maintained by the tuple space. Before expiration a producer (or some other entity) is responsible for the *renewal* of a lease. Otherwise, after expiration of a lease, the garbage collector process is free to delete a tuple without further notice. Thereby, a simple but effective means of garbage collection is established.

However, JavaSpaces considered as a whole is comparable to TSpaces. It has almost the same strengths as well as weaknesses. In the most basic case it is cetralized, federation is not explicitly supported, although possible, and disconnection can result in fault situations not desired. Moreover, the impact on client resources is considerable and often makes the use in mobile settings impossible. This is true for two reasons: first, a complete Java2 environment is needed; and second, the underlying Jini infrastructure makes heavily use of *mobile code*, i.e., a large set of classes has to be downloaded, inspected, and instantiated at runtime on a device. This introduces requirements for computational resources, as well as bandwidth, usually not met by resource-limited devices. On the other hand, according to our own experience, JavaSpaces is probably the most flexible and stable tuple space available.

L2imbo. L2imbo [DFWB98b; DFWB98a] is a tuple based infrastructure with the emphasis on quality of service. It supports: multiple tuple spaces, tuple type hierachies, and quality of service attributes, thereby emphasizing its main focus. Like TSpaces and Lime new tuple spaces can be

created on the fly when needed, although L2imbo hereby relies on a central instance of a main tuple space. This is somehow comparable to the "Galaxy" in TSpaces. The tuple spaces are implemented in a distributed fashion such that each host holds its own replica of the tuple space. This allows for disconnected operation but seriously impairs scalability. Tuples can be assigned quality of service attributes, like a *deadline*, indicating the lifetime of a tuple in the space and comparable to the leasing mechanism in JavaSpaces. Maintaining consistency and meeting certain QoS specifications is left to a number of agents, responsible for monitoring the overall system, the creation of new tuple spaces where needed, and the propagation of tuples between tuple spaces. L2imbo is probably the most advanced system for the use in mobile settings, although explicit support for location- and/or context-awareness is missing.

3.5.3 Multicast and Geocast

Multicast offers one-to-many communication that transmits a single message to potentially multiple receivers. A thorough overview[7] can be found in [WZ99]. Originally, multicast was incorporated into network level protocols, in particular into the Internet Protocol [Dee89]. More recently, multicast is considered on the application level [hCRSZ02; RHKS01]. Deering and Cheriton [CD85] provide a good introduction to multicast in the context of wide area networks (WAN). Multicast concentrates on the efficient sending of messages to a specific group of receivers, the so called multicast groups. In IP multicast these groups are separate sets of receivers. Neither explicit relationship between multicast groups are specified, nor does IP multicast provide means for such specifications. Hence, structuring applications on the basis of multicast groups delegates the specification of relationships and their management on the applications. Also problematic is the namespace of multicast groups: routing is performed on group identifiers alone. Semantic meaning of content which is published and consumed by a group of multicast clients is either determined by the actual multicast group or its position relative to a "subject tree," as defined in Section 3.2.3. However, multicast is sufficient for many application scenarios, although it often suffers from the lack of support and availability of the Internet Multicast Backbone (MBONE). Nevertheless, for small to medium sized networks under a single administrative domain, multicast can be sucessfully deployed. For instance, REBECA uses multicast for discovery of peer routers based on the Java Naming and Directory Interface (JNDI) [Inc01]. Another example is the Jini architecture, where the process of service discovery and federation also is based on multicast groups [Sun99b].

Geocast [NI97; RFC2009] can be seen as a specialized form of multicast, where availabe positioning systems for mobile devices are used to determine the actual location for location-dependent delivery of messages. Thereby, the underlying semantics for filtering and addressing is similar to multicast. Hence, geocast can be classified as a special variant of multicast. In multicast, a message is sent to a group of receivers. Either a receiver explicitly joins a certain multicast group or a group is defined implicitly, e.g., through the geographic position of a mobile device. This is the exact semantics of geocast: the implicit generation of multicast groups based on the location of mobile clients and the target area of messages.

The geocast implementation presented in [NI97] distinguishes the following components:

Geocast clients. A geocast client usually is a software component, for instance a loadable software library, running on a mobile device. Its responsibility is to send messages received from local applications to a *geocast router* for forwarding.

[7] At least for German-speaking readers.

Geocast routers. Routers are responsible for the forwarding of messages from the sender to the clients. Common message forwarding mechanisms are (i) *geographic routing*, (ii) the *directory-based* approach, or (iii) the *multicast-based* approach. Approach (i) uses geographic information directly for message forwarding, i.e., special routers compare the target area of the message and the areas covered by sub-networks to decide where a message must be forwarded to. This approach obviously requires a strong ordering of the network, together with specialized routers, because the structure of the network has to correspond to the underlying physical layout. Approach (ii) is IP-based in the sense that all (potential) receipients register with a directory service, i.e., their IP-addresses together with the current location. The directory service then generates mappings between geographic target areas of messages and the associated IP-addresses. Thereby, implicit multicast groups are generated. However, the actual update process of moving objects in the directory remains unclear. Approach (iii), finally, is based on a "traditional" multicasting approach. The physical space is divided into multicast-addresses, i.e., some partition exists which makes a certain target area for a geocast message addressable by a unique mulitcast-address. How such partitions can be obtained and how an appropriate multicast address is determined in a dynamic environment remains unclear, too. A proposed simplified solution for the determination of a target address is to replace multicasting by broadcasting and client-side filtering.

Ideas for using geocast in the automotive sector are presented, e.g, in [MFE03; ME04]. Also, STEAM [MC02], a middleware service designed for wireless local area networks using the ad-hoc network model where there are no access points and system wide services, can be seen as an implementation of geocast. Subscribers only consume events produced by geographically close-by publishers. For this it relies on proximity-based group communication [MKCC01]. Another system using geocast is Nexus which we will describe below in Section 3.5.4.3.

3.5.4 Selected Work from Pervasive Environments

Many research projects exist in the field of ubiquitous and pervasive computing. In this section we want to restrict ourselves to those projects that we feel have a strong inclination towards infrastructures and middleware, as well as some background in the use of an event-driven communication paradigm or event service.

3.5.4.1 CALAIS

CALAIS is a system architecture for the support of context-aware applications. In his Ph.D. thesis [Nel98], Nelson proposes an event-based model for the tracking, extraction, and management of location- and context-information. The actual distribution of such information is based on a centralized CORBA event service. The system exploits the abstraction capabilities of the OMG interface definition language in order to provide an abstraction layer from the actual sensor technology in use. Each sensor is described in terms of a CORBA interface, its attributes, and event classes it can produce. In a certain sense, type-based filtering is used, based on the interface of the sensor abstraction. The main drawback is that the CORBA event service provides a rich set of functionality on the one hand, but clearly lacks support for small resource limited devices. Moreover, scalability issues remain unclear under the assumption of a purely cetralized solution.

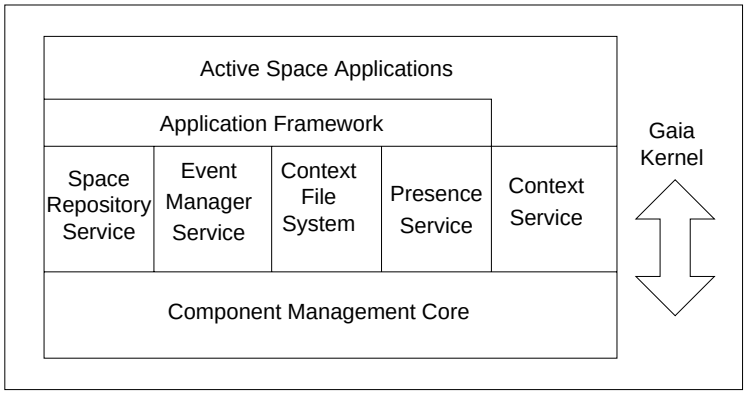


Figure 3.3: Gaia architecture.

3.5.4.2 Gaia

The goal of the Gaia project at the University of Illinois at Urbana-Champaign [RHC+02] is to create a complete meta-operating system for the use in pervasive computing environments. In [RHC+02] the motivation for this project is described as "the lack of a suitable software infrastructure to assist us in the development of applications for ubiquitous computing habitats or living spaces." The focus of Gaia therefore are so called *Active Spaces*, i.e., homes, offices, and meeting rooms enriched with embedded "smart" artifacts. Gaia tries to bring the functionality of an operating system to physical spaces, thus, it is also referred to as Gaia OS. Operating system functions, such as events, signals, file system support, security, processes and their management should be supported by the overall *meta-operating system*. Moreover, typical operating system functions are made "context-aware" and many of the functions are tailored for Active Spaces. It provides services for location, context, and events, and repositories with meta-information about the current active space.

The three major building blocks of Gaia are: the Gaia kernel, the Gaia application framework, and the Gaia applications (Figure 3.3). This constitutes a distributed middleware infrastructure that coordinates software entities and heterogeneous networked devices in a physical space. Gaia OS is influenced by previous work on *2K* [KCM+00; KSC+98] a reflective middleware [KCBC02] operating system build on top of a traditional OS. The underlying communication infrastructure is based on TAO [SLM98; KRL+00] and CORBA. Interesting in the context of this thesis is the event manager service that is based on the CORBA event service. It is responsible for event distribution in the active space. The addressing scheme is *channel-based* as introduced in Section 3.2.3. Each channel has one or more producers and one or more consumers attached. The event manager service itself is centralized and provides a single entry point to the event mechanism. Gaia OS mainly uses the event mechanism for signaling purposes, like distribution of information about new services, applications, people, errors, and component heartbeat. On the other hand, application or component interaction is based on direct, RPC-like communication mechanisms.

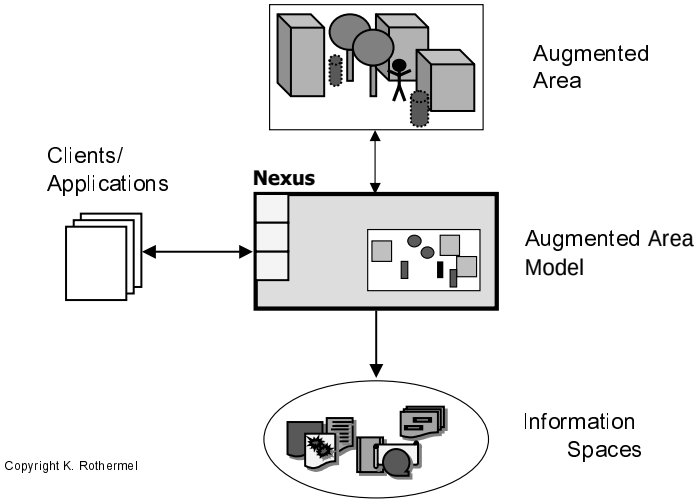


Figure 3.4: Overview of the Nexus architecture

3.5.4.3 Nexus

Nexus is a massive effort to create a "digital world model." Information about real objects is embedded into a larger virtual model of physical space. By connecting information about physical artifacts and the surrounding environment with additional "purely" virtual information, novel applications and services are expected to emerge. The whole project is centered around *augmented area models* [HKL+99b]. An augmented area model is the virtual representation of the physical space together with embedded information about clients, services, and additional information. The augmented area model allows Nexus-based applications to benefit from the users' position as well as the relationships to other objects around them.

Every change observed by the Nexus platform is automatically propagated to the model and might trigger actions in the augmented area model. In this respect, asynchronous communication and notifications are necessary. As a complete description of Nexus clearly is beyond the scope of this thesis, we refer to the literature for an overview (e.g., [HKL+99a; NM01]).

In [BR02] the authors describe the idea of spatial events, distinguishing between *basic events* and *complex events*, i.e., composite events. They decribe the most basic components of an event specification language. Details about the actual set-up and integration into the Nexus platform are omitted in the literature.

Related to the model we introduce in Chapter 4 is the work presented in [DR03] and partially in Bauer et al [BBR02]. There, the authors describe a location model for the use together with geocast as introduced in Section 3.5.3 above. The location model the authors propose clearly is related to other models, like [Leo98; KEG93; EF91; JS02]. The proposed model uses a hybrid model of location specification. Geometric as well as symbolic location specification is allowed. The model presented allows for a fine-grained specification of target areas as introduced in the geocast communication paradigm. However, location specification is done in XML and basically follows its predecessors in using, for instance, hierarchical symbolic location descriptors for a target area,

like /de/berlin/keplerstr/9/floor1/wing1/room72. As common in hybrid location models, conversion functions for translating symbolic into geometric locations, and vice versa, are provided.

3.5.4.4 SOLAR

The SOLAR [CK01; CK02b; CK02a] research prototype was developed at Dartmouth College. Its main focus is to provide abstractions for the support of adaptive ubiquitous computing applications. They identify the key challenges as being context collection, aggregation, and dissemination. Applications must be allowed to flexibly define their own operations and their composition from "lower-level" operations. The SOLAR model is based on a *directed acyclic graph* (DAG) of operators. The underlying assumption is that context-driven applications adapt to context changes in an event-driven fashion. Therefore, valid input to applications can be characterized as an event on a high semantic level. Such input must be generated out of low-level information as it is provided by sensors. The sensors in their model are called *event sources*. Each event source produces an *event stream* over time, which is considered unidirectional. An event consumer subscribes to such event streams. The main contribution is the definition of a DAG consisting of *operators*. Operators are subscribers to data and consume one or more event streams and produce new events based on the actual incoming data. Operators, obviously, can be recursivly connected to form an operator graph structure. Operators are deterministic functions comparable to event compositors as introduced by CEA, HERMES, or in [LCB99; Cil02]. Subscriptions are type-based, name/value-based, or an XML document. However, it remains unclear how these different types of subscriptions can be made comparable, or whether they represent separate concerns of a system. This is especially true for the lack of a common location model for use within the infrastructure. Although the SOLAR architecture defines the notion of location-aware subscriptions, apparently they are not exploited for efficient matching of events.

3.5.4.5 one.world

one.world [GDH+01; Gri02] is a system architecture for pervasive applications developed at the University of Washington. The main focus is a complete integrated framework going beyond the scope of this thesis, namely to provide a whole suite of services for application development and deployment in a pervasive computing environment. The overall architecture is shown in Figure 3.5. The underlying assumption for their work is that comtemporary systems try to hide distribution and rely on technologies such as RPC [RFC1057] or distributed file systems [LS90] and therefore failures are hard to anticipate. Such systems try to extend single node programming mechanisms, tight coupling, and encapsulation of data and functionality to the pervasive computing case. Consequently, applications on top of this functionality tend to be structured like single node applications and therefore are likely to fail.

The one.world project approaches these issues by making distribution explicitly accessible to applications. Basically, the system tries to cover three main requirements: exposure of contextual changes, ad hoc composition, and sharing of information as a default, especially in collaborative scenarios. In our oppinion the work presented has much in common with the goals and the design of Jini [Sun99a].

However, as it can be seen in Figure 3.5, notification delivery, asynchronous communication, and filtering is only one of the services specified. Thus, the focus lies elsewhere, e.g., service discovery and data management (for instance *checkpointing* or information sharing via tuples).

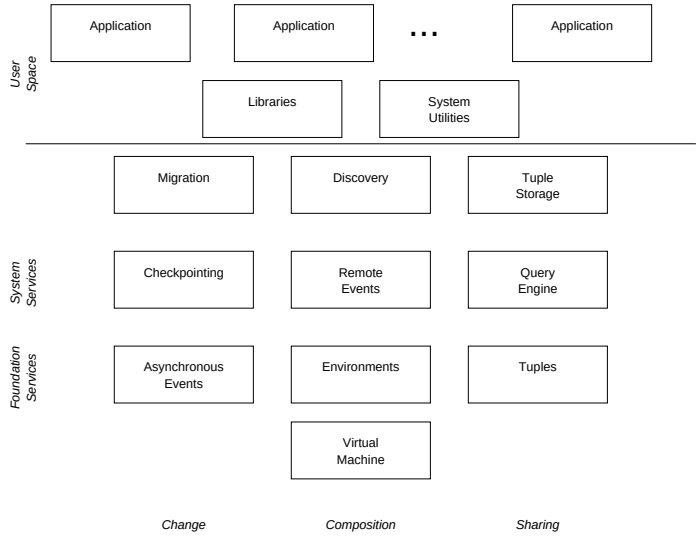


Figure 3.5: Overview of the one.world architecture

3.6 Summary

In this chapter we laid the foundations of publish/subscribe for readers not familiar with this important paradigm. As a second step, we introduced REBECA: an implementation of the aforementioned publish/subscribe mechanisms. REBECA is a distributed publish/subscribe notification service and serves as basis for all extensions we present in the remainder of the thesis. Extending REBECA is necessary because its original focus is on scalability, manageability, and the exploration of efficient routing mechanisms. Therefore, important aspects of mobile systems are not considered in its architecture and need to be integrated as done in this thesis. We analyzed REBECA and compared it against the requirements for infrastructure support we identified in Section 2.2.4. We clearly stated where REBECA, and the publish/subscribe paradigm in general, fall short for effective support of mobile clients in pervasive environments. The main shortcomings are (i) proper support for mobility, (ii) context-awareness as eligible system property for roaming clients together with mechanisms to delegate certain aspects of context-handling to the infrastructure, and (iii) adaptation of the publish/subscribe model to decouple producers and consumers in time and space efficiently. Provision of solutions to address these shortcomings is the main focus of the remainder of this thesis. Finally, in this chapter, we introduced related work and extensively analyzed projects relevant for this thesis. In order to structure the discussion more clearly, we subdivided it into relevant work done in the field of publish/subscribe systems and work relevant in the context of pervasive computing infrastructures.

4 Foundations of Context and Location for Publish/Subscribe Middleware

When we try to pick out anything, by itself,
we find it hitched to everything else in the universe.

John Muir, naturalist and conservationist (1838-1914)

4.1 Introduction

In Section 2.2.3 we gave the broad view on *context* and the different categorizations of *context-awareness* as they are common in literature. *Context-awareness* often is used synonymously for changing the behavior of applications corresponding to changes in the surrounding.

In this chapter we argue that context-awareness is also an issue for a publish/subscribe notification service. We specify a model of context in a way that it is applicable for the operation of such middleware. In order to do so, we have to concretize the broad view of context as introduced in Chapter 2. The focus of our discussion about context-awareness is aiming towards an operational definition as one central foundation to be used in the remainder of this thesis.

Central to the discussion is the role of *location* as a primary source for adaptive behavior of applications. Therefore, we specify and then integrate a formal model of location as first-class abstraction into the very core of the publish/subscribe notification service. The challenge here lies in the nature of location specification and how it can be mapped to the model of content-based filtering we assume as underlying filter model.

This chapter is structured as follows: after specifying what we understand as context for the use in a notification service in Section 4.2, we introduce and discuss the design dimensions for location-awareness as a first-class object within a publish/subscribe infrastructure in Section 4.3. The realization of the concepts introduced in the following sections is the central topic of Chapter 6. Throughout the remainder of this chapter we will analyze the requirements for a proper specification of a location model. Then, we review common solutions as they are used in location-aware systems to clarify requirements stemming from the application level. Finally, based on the previous discussion we propose a specific location model to extend the underlying REBECA notification service.

4.2 Modeling Context for Publish/Subscribe Infrastructures

As we have shown in Section 2.2.3.1 and especially in our discussion about categories of context definitions in Section 2.2.3.2, rather diverse views of this topic are common in the related literature. They range from simple enumerative or declarative definitions over somehow operational definitions

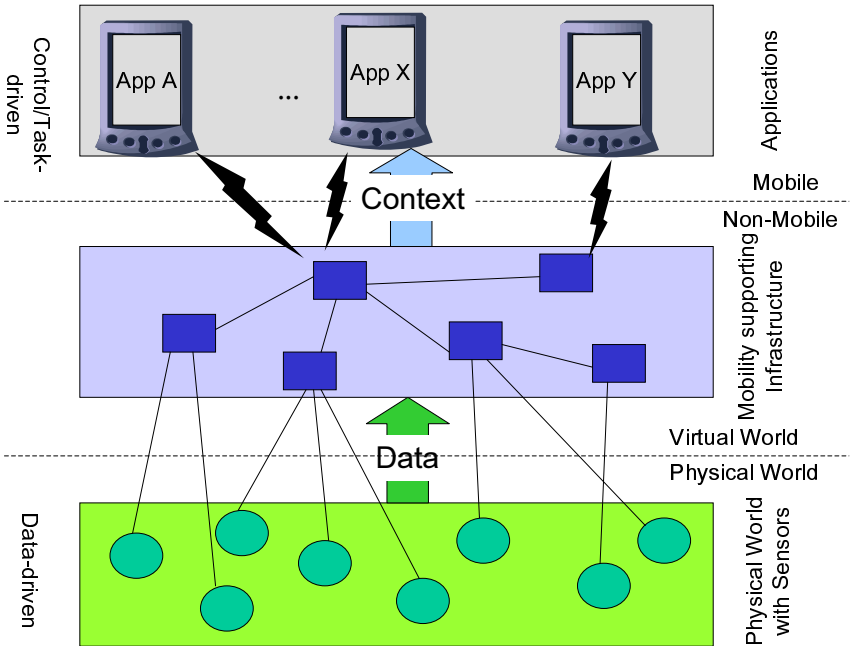


Figure 4.1: General model: data-driven sensors, a nomadic, event-driven infrastructure, and applications on mobile devices

to concrete categorizations of the term *context*. But context often is considered to be a concern unique to applications.

For this thesis, we take a different point of view and define *context* as a concern of a distributed publish/subscribe notification service. Its basic defining property therefore has to be whether context can be exploited to optimize or at least influence the operation of the publish/subscribe infrastructure. This is legitimate because, in a general sense, context is any information contributing to the adaptation of applications. Obviously, the same argument shall be applied to the intermediary infrastructure.

Hence, our definition of *context* is:

DEFINITION 4.2.1 **Context.**
Any information that can be exploited to influence the operation of the distributed publish/subscribe notification service is considered to be context.

In other words, whenever information can be used to influence the routing decision made in the infrastructure, it is considered context. This is particularly benefitial whenever the routing decision is optimized by exploiting context information.

4.2.1 Categorization of Context Information

Context is an umbrella term for any data about the surrounding as a whole. *Context information*, on the other hand, describes the particular building blocks *context* is comprised of on a finer-grained level. In this thesis, *basic context information* is assumed as being data that is collected or computed based on raw-data from sensors, building up the application's context (or situation) as defined above.

Two kinds of context information can be distinguished: (i) application-specific and (ii) system-specific context. Application-specific context uses semantics intrinsic to a specific application (domain). System-specific context exploits data that is defined or generated outside of a concrete application. Usually, such context information is usable within the notification service for routing optimizations.

Application domain-specific context. Any context information which is interpretable only by a concrete application or application domain is categorized as application domain-specific context. For example, assume that a temperature sensor periodically generates temperature readings, like "((temperature,25)&&(unit,C))", and injects them into the system using the publish/subscribe infrastructure. On the one hand, a matching application may use such information, based on the implicit sematics used. On the other hand, such information could only be used for routing decisions if an appropriate ontology is deployed and used in every router within the routing network. Other examples include computed context descriptions, like "health-state: OK" (cf. Chapter 8.3). They refer to some application's state and thus cannot serve as *general* context information.

System-specific context. Complementary, system-specific context is any information which is exploitable as additional source of information about the current system's setting. This is information which is valuable outside the content of a message. The semantics of the information must be made explicit and is agreed on by every part of the broker network. Examples for such information are: location, time of day, identifiers, credentials, or, in general, any other system property.

However, for any context information being usable as system-specific context, the following properties must hold:

1. An explicit and well-defined data model exists.
2. A matching filter model exists, i.e., a mapping from data model to a boolean value must be possible at all times (see also the following item).
3. An explicit metric on the data model is available to ensure comparability of any two instances of the data model. This ensures the completeness of the previous item.

Although trivial at first sight, the previous enumeration is crucial for the definition of *context* for a publish/subscribe notification service. The first item simply states the need for a well-defined and *explicit* data model. A common agreement on the semantics of the data model must exist. Item 2. ensures that the *routing decision* can be computed. This includes the implicit requirement to do so efficiently. Item 3. makes explicit that only well-defined context models are eligible as system-specific context.

One striking example for eligible context is *location*, as we will show throughout the next sections. *Not* well-suited for the use in routing decision are "fuzzy" terms, like *situation*. Here, the open questions are (for example): how are two *situations* comparable? What is their metric? What distinguishes one situation from another?

Before defining location as context information, next, we incorporate the notion of context into our definitions of filters and notifications.

Context-dependent filters and notifications. For completeness, we define:

DEFINITION 4.2.2 **Context-dependent Filters.**
A context-dependent filter is defined as an extension to the standard data and filter model of a notification service. In addition to the standard query specification a filter offers means to specify metadata, possibly influencing the routing decision.

DEFINITION 4.2.3 **Context-dependent Notifications.**
A context-dependent notification is defined as an extension to the standard notification model of a notification service. In addition to the standard content transported, metadata can be specified, concretising the context of a data item.

4.3 Modeling Location for Publish/Subscribe Infrastructures

The focus on mobile users and mobile devices naturally makes location an important issue to tackle. For example, in *Location-based Services* (LBS) information is related to the actual position the LBS currently is deployed. Here, typical examples include: local weather forecasts, traffic conditions for the current area, or restaurant menus in the vicinity. Applications might want to issue subscriptions in a way that the subscription's *relevancy* is restricted to a certain area in the real world. This is what we have introduced as *location-dependency*. Relative to a certain area a data item is valid, and hence is forwarded, or is invalid and therefore discarded.

As a guiding example we take a subscription to a "Free Parking Spaces" service. On the application level the goal is to be informed about available parking spaces in the vicinity. For the sake of simplicity, we formulate this as an informal subscription: "subscribe to *Free Parking Spaces Service* and around a radius of 0.5 kilometers around my current position"

This subscription has two distinct parts: (a) an application- and domain-specific part, identifying the application-dependent domain knowledge ("Free Parking Spaces Service"), and (b) a well-known and well-defined description of the "range" in which this subscription should be valid. This adheres to the definition of a context-dependent filter as introduced in the last section. Furthermore, notifications matching this particular subscription also consist of two separated parts.

- *Content.* Following the standard model of REBECAwhich uses content-based addressing, the whole content of a notification is eligible as matching criteria for subscriptions and notification. The same semantics is maintained here. The issuer of a subscription expresses interest in information concerning the shared knowledge about some application domain, e.g., the knowledge about the status of parking spaces in the current area.

- *Envelope.* By delegating knowledge about the location into the envelope of a notification, a separation of concerns is introduced. Therefore, consumers and producers can rely on a well-defined interface to specify location information, while the infrastructure can use the same model of location to optimize routing and relieve location-aware applications from (a) redefining location models for each application separately and (b) receiving information not

relevant for reaching the application's goal by specification of validity constrains on notification delivery.

4.3.1 Design Space of Location Models for Publish/Subscribe

When exploring the design space for a location model for publish/subscribe systems, the main challenges are:

- Finding a data model suitable to accommodate the intended use of "location" as an application and infrastructural concept at the same time. Obviously, some models are quite natural for the use in a specific application or application domain. For example, the use of floorplans is intuitive for modeling indoor applications, but not applicable in outdoor scenarios. Hence, from the middleware perspective, we are looking for a versatile model supporting a large variety of applications.
- Operations defined on the data model are needed to facilitate location-awareness for applications and efficiently matching filters and notifications in the middleware at the same time. This includes the specification of an appropriate definition- and query language for location-aware subscriptions and notifications, respectively.

4.3.1.1 On Location and Space

In this thesis, we focus on publish/subscribe middleware that supports devices roaming in the physical space. To specify proper mechanisms for such support, we need a deeper understanding of location and space. In general, the main distinction between "location" and "space" is that every location is part of a larger space. This space might contain other entities, devices and users. From an application's perspective, the relationships between those are important. We distinguish [1]:

- *Location in space.* A device always is located at a physical location within a certain space.
- *Interaction.* Spaces might contain more than one object. A device might want to interact with other devices (e.g., sensors) or users.
- *Reactive behavior.* Devices are situated within a space and therefore are subject to changes, triggered by influencing events from the space, or from other devices and users within it.

Essentially, while roaming devices are embedded into changing spaces and interact with the current space or other entities in the same space mediated by the middleware deployed there (cf. Figure 2.3 on Page 12). This is the basic application of the *nomadic system model*.

Additionally, for the design and implementation of a publish/subscribe infrastructure for pervasive systems it is important to understand the basic model of spatial interaction:

1. Location of a device.
2. Mobility through space.
3. Interaction with other entities in the same space.
4. Application-specific awareness of other devices.

[1] The characterization presented here is inspired by a similar characterization in Dix et al [DRD+00].

Unfortunately, built-in middleware support for the items above can only be realized to a certain degree. We have to consider the actual movement of a device in space for efficient and complete information delivery. Moreover, support for the interaction with changing partners and some means to support context and location-awareness should be built into the middleware. However, support for application-specific awareness naturally seems to be out of scope of a general purpose middleware approach.

4.3.2 A Taxonomy of Location Models

In general, we can distinguish between two basic classes of location models:

- Geometric models.
- Symbolic models.

Some applications make use of the exact location of a device, where the location usually is provided as Cartesian coordinates, as they are provided, for example, by the Global Positioning System. Those systems use a *geometric model* of space. For other applications, like tour guides or office applications, the location is expressed in terms of a (hierarchical) topology, which is sufficient for understanding the location of a device and its relationship to other entities. Usually, in such a system location is expressed in terms of a unique canonical name in a well-structured namespace, mapping a physical location to a unique name. Such symbolic representations of location are called *symbolic models*. A prominent example for this use is the cooltown project, where location can be expressed in terms of a URL, referencing to a web-page associated with a certain location.

In general, it is necessary to model not only the actual location of entities but also some of the relationships between them. For example, common "questions" asked by context-aware applications are:

(i) "Where am I?",

(ii) "What else is nearby?", and

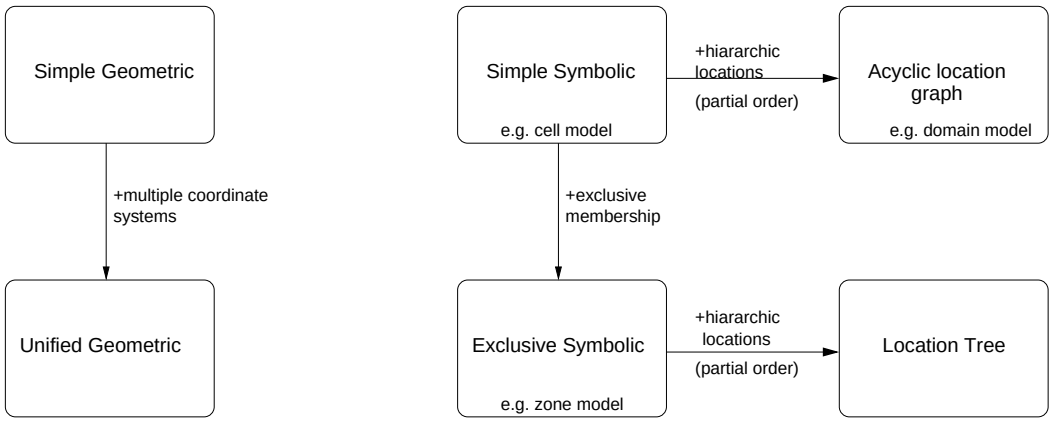
(iii) "How should I behave in the light of (i) and (ii)?" (cf. [DRD+00]).

Here, we concentrate on a model for (i) and (ii) because (iii) is highly application-dependent and therefore not a direct concern of a middleware solution. In practice, an application always accesses "location" as a conceptional representation and not "directly" in the physical world. Hence, for the specification of context and location, we need:

- A *semantic model* of space and location which offers a well-defined *data-model*.
- A *computational model* of space and location being part of the puplish/subscribe infrastructure and serving as a basis for a well-defined *filter-model* within REBECA.

A filter model for a publish/subscribe middleware should make use of the semantic model for the definition of *relationships* between space and location as foundation for the matching of filters and notifications based on, e.g., *containment* or *intersection*.

Now, we will analyze geometric and symbolic models in the light of the requirements for suitable data and filter models.



(a) Geometric models (b) Symbolic models

Figure 4.2: Classification of location models

4.3.2.1 Geometric models

An intuitive model of location is the familiar (x,y,z)-coordinate model we know as a simple geometric model of space (cf. Fig.4.2(a) and [Leo98]). All geometric models have in common that they are based on some reference coordinate systems. A location then can be expressed by a *set of coordinate tuples*, describing a position relative to the reference coordinate system of choice. Interestingly, there is no explicit distinction between location and located object in such models as everything is simply a set of coordinate tuples. Hence, the main challenge in modeling location using a geometric model lies in introducing an explicit destinction between location and located object, i.e., entities and the space they are located in. An important step to make coordinates useful for location representation is to define "location abstractions" or "location symbols" on top of the pure coordinates. The obvious abstractions are boxes (2D or 3D) or more complex shapes which are modeled on top of coordinates. Together with appropriate measurements of "nearness", it is then feasible to differentiate between a location of an object and the space it is contained in. The central advantages of geometric models are their flexibility and versatility. Coordinate-based spatial models provide very flexible means to access location information. Due to their simplicity a geometric model is easily implementable in sensors and applications. An obvious disadvantage is the lack of structure and missing hierarchical layering of location information, making system layout and design more complicated.

4.3.2.2 Symbolic Models

Symbolic models refer to locations by abstract symbols, although the most common case is to use *named locations* in a human understandable way. The symbols are characters and names, like "Wilhelminenstr. 7", "Room C120", "Andreas' Office", etc. The main characteristic of symbolic models is that locations and located objects are represented by symbols that adhere to natural *ordering criteria*, i.e., sets and located objects as members of sets, respectively. In the physical space this means,

whenever a located object is member of a location set, it is physically within the range of this particular location, e.g., "(user::andreas) in (office::andreas)" might evalute to $true$ whenever a uniquely identifiable user named "Andreas" is in the office associated with the same user. Figure 4.4(b) shows the classification of symbolic models as introduced by Leonhardt in [Leo98] and which is the basis of this section. The most basic models are simple symbolic models which may be constrained, e.g., by not allowing spaces to overlap. Equally, the model of space and location might include a partial ordering of location symbols based on the spacial inclusion of the underlying spaces. This results in different (specialized) models of space and location, as we will explain later in this chapter. In comparison to geometric models the prime advantage of symbolic models is that they mostly are hierarchical, discrete, and well-structured, facilitating naming, adaptation, and scalability. On the other hand, they are more complicated to handle and, most important, usually are subject to application-dependent naming and semantics. Therefore, versatility is strongly limited. In general, sensors and applications from different domains would have to rely on ontological translations for interoperability which is not feasible at all times.

4.3.3 A Geometric Model of Space

The predominant (outdoor) positioning system today is the Global Positioning System (GPS) which is based on the World Geodetic System (WGS84) [WGS84] reference system, i.e., longitude, latitude, and altitude. The WGS84 adequately models location by itself such that we can concentrate on structuring the flat space of (x, y, z) tuples with symbolic spaces on top of the coordinate abstraction.

With respect to the geometrical model we have to model two abstractions, location and a space. Location is ideally modeled as a point or, as no positioning system is error-free, as *uncertainty area* in which it is likely that the entity is contained. However, located entities are always contained in a larger space. Therefore, it is important to model certain relationships between location, located object and space, which can be handled uniformly by assigning "space", i.e., a spatial extension, to every located entity and space.

Once a data-model of location is established, we can introduce a computational model of location where relationships between locations and spaces can be expressed, i.e., inclusion, overlap, etc. These relationships then can be used to specify constraints for the delivery of notifications to specific clients, based on location-dependent subscriptions issued (cf. Chapter 6). However, tests on spacial constraints are non-trivial when we allow the use of complex shapes. Thus, we will weaken the notion of location and space by introducing and using a *location graph* for the efficient matching of filters and notifications within the infrastructure. This is discussed in Section 4.4.2.

Specification. The specification of location and located objects presented here is based on similar specifications found in the literature, especially in [Leo98].

For the sake of simplicity we constrain the three-dimensional notion of *space* to the two-dimensional notion of *area*. The definitions presented here are easily extensible to the third dimension, which is omitted here. We define:

DEFINITION 4.3.1 **Area.**
An Area is any geometrical object with a spatial extension.

Based on the this definition each *location* can be modeled as a small Area[2], which is in accordance with a non-error free model of location measurement. Further, we can model the relationships

[2] Please note that this also applies to a point.

between locations based on Area. For the intended use as computational model being part of the publish/subscribe middleware, we consider overlap and inclusion as primary relationships. Please note that we model the operations to be *decision functions*: they directly map their input to a boolean value. This way, they are directly implementable in the event broker network.

We specify:

DEFINITION 4.3.2 **Geometric Location Model.**

$$\begin{aligned}
\texttt{anyLoc} &: Area \\
\texttt{contains} &: (Area, Area) \rightarrow Boolean \\
\texttt{overlaps} &: (Area, Area) \rightarrow Boolean
\end{aligned}$$

$$\begin{aligned}
\forall a_1, a_2 \in Area &: \texttt{contains}(a_1, a_2) = true \\
&\Rightarrow \texttt{contains}(a_2, a_1) = false \quad \text{(asymmetry)} && (4.1) \\
\forall a_1, a_2, a_3 \in Area &: \texttt{contains}(a_1, a_2) = true \wedge \texttt{contains}(a_2, a_3) = true \\
&\Rightarrow \texttt{contains}(a_1, a_3) = true \quad \text{(transitivity)} && (4.2) \\
\forall a \in Area &: \texttt{contains}(\texttt{anyLoc}, a) = true && (4.3) \\
\forall a_1, a_2 \in Area &: \texttt{overlaps}(a_1, a_2) = true \\
&\Leftrightarrow (\exists a_3 \in Area : \\
&\texttt{contains}(a_1, a_3) = true \wedge \texttt{contains}(a_2, a_3) = true) && (4.4)
\end{aligned}$$

Whereby `anyLoc` is a special function, serving as a "wildcard" for the specification of location information and evaluates to $true$ for any location specified. The inclusion relationship is transitive and asymmetric.

Complementary to `anyLoc`, we specify an additional placeholder for the specification of the current location of a located object.

DEFINITION 4.3.3 **Geometric Location Reference.**

$$\texttt{myLoc} : ObjectId \rightarrow Area$$

`myLoc` returns the location of an entity using a system specific identifier.

4.3.4 Design Space of Symbolic Models

In the relevant literature, especially in [Leo98], three models of symbolic locations are distinguished (cf. also 4.4(b)):

- The basic *cell model*;
- The more advanced *zones model*;
- The most flexible *location domain model*.

As the *cell*- and the *zones model* are of lesser importance for this thesis, for those models, we will give only a very brief characterization before we introduce the *location domain model* in more detail.

4.3.4.1 The Cell Model

The *cell model* is characterized by an intermix of different location sensing technology, leading to a significant lack of structure within the location model. Each location sensing technology is defining its own location model in terms of a well-defined geographical area, e.g., a room, a bluetooth cell, a wireless LAN cell, etc. As those areas are not exclusively assigned to a single location system, a located-object might be detected by several different location systems, leading to the loss of bijective mappings between locations and located objects. However, each of such cells is a separate *symbolic location* in this model. They are allowed to overlap, but an inclusion relationship among cells is not part of this model.

4.3.4.2 The Zone Model

The *zone model* is an extension to the cell model and is mainly characterized by the structure added compared to the former model. The main distinction from the cell model is that, given the same intermix of sensing technology (i.e. overlapping cells in the real world), the "logical" map is free from overlapping cells (cf. Figure 4.5(c) on Page 67). This is done by superimposing new *zones* on the actual cells. The intersections between cells then are represented by new zones, making the model free of overlapping. The zone model can be classified as *exclusive symbolic model*, as any located-object can only be part of a single zone.

Please note that exclusive symbolic models in general are of great interest as within an *exclusive* symbolic location space the movement of a single object can be modeled in terms of graphs and finite state machines, which is naturally given by the overlap-free zone model. In [Leo98] Leonhardt states: "*Hence, a zone space is a natural framework for persistent object tracking and movement prediction*".

4.3.4.3 The Location Domain Model

The *location domain model* is a formalization of an intuitive location understanding introduced in [Leo98] and can be found in its various forms in other location-aware systems. The basic idea is to define *location domains* as symbolic locations that can be ordered with respect to other location domains. Therefore, the "contains" relationship is instrumentalized as a partial order on location domains. As the underlying geographical areas also are "contained" in each other, this model is understandable intuitively.

A location domain corresponds directly to the spatial relationships found in the real world: e.g., a *floor* is subdivided into different *rooms* and itself is part of a *building* which itself belongs to a larger organization, like a *university*. Hence, location domains in the location domain model form a *location tree* or -*lattice*, i.e., a directed *location graph* (cf. Fig. 4.3). Please note that usually the "contains" relationship gets application specific and models a specific *view* on a location graph. In Figure 4.3 we have depicted this by dividing the location graph into a geographical "physical layout" and an application specific "logical layout".

Obviously, to allow for strong consistency within the model any located entity that is member of a location domain must also be member of all parent location domains. Mobile objects also can be modeled as mobile location domains. The implication is that the position of a mobile domain within the inclusion domain changes over time.

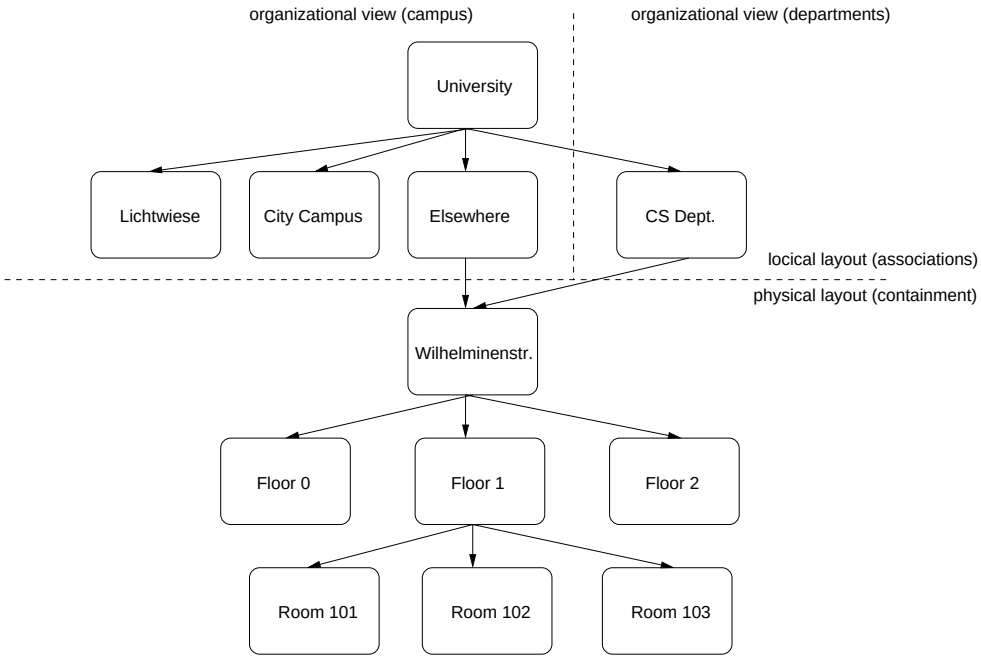


Figure 4.3: The location domain model

Specification. As the location domain model shall build the interface between applications and the underlying publish/subscribe middleware we include a more detailed specification of the properties briefly discussed above.

For the specification of a location domain model, we informally introduce two abstract data-types: Location and Entity.

DEFINITION 4.3.4 **Location**
A Location is a unique identifier referencing a certain spacial extension.

DEFINITION 4.3.5 **Entity**
An entity is uniquely addressable in a location domain, e.g., a mobile device.

Based on the Location definition above and the (implied) structure as shown in Figure 4.3, it is possible to define a location hierarchy. To have an operational description, we define two operations on Location: `contains` and `disjunct`. The `contains` operation resembles the same operation in the geometrical model but, additionally, is also irreflexive. Obviously, the operation is asymmetric, as no spatial inclusion can work both ways at the same time. A room can be part of a floor but not the floor part of the same room. We added irreflexivity, such that no location is part of itself. As we require locations also to be contained by all its parents, `contains` is transitive. By modeling `contains` this way we induce a partial ordering on locations and thus a location hierarchy.

Opposed to the geometrical model, we introduce a function to decide whether two locations are disjunct. In the “flat” geometrical model it was important to add structure to the model by identifying

overlapping areas. As the location model used here already has structure, it is more important to know which locations do *not* overlap.

DEFINITION 4.3.6 **Symbolic Location Hierarchy**

$$\begin{aligned}
\texttt{anyLoc} &: Location \\
\texttt{contains} &: (Location, Location) \rightarrow Boolean \\
\texttt{disjunct} &: (Location, Location) \rightarrow Boolean
\end{aligned}$$

$$\begin{aligned}
\forall l_1, l_2 \in Location &: \texttt{contains}(l_1, l_2) = true \\
&\Rightarrow \texttt{contains}(l_2, l_1) = false \quad \text{(asymmetry)} && (4.5) \\
\forall l_1, l_2, l_3 \in Location &: \texttt{contains}(l_1, l_2) = true \wedge \texttt{contains}(l_2, l_3) = true \\
&\Rightarrow \texttt{contains}(l_1, l_3) = true \quad \text{(transitivity)} && (4.6) \\
\forall l \in Location &: \texttt{contains}(l, l) = false \quad \text{(irreflexivity)} && (4.7) \\
\forall l \in Location &: \texttt{contains}(\texttt{anyLoc}, l) = true && (4.8) \\
\forall l_1, l_2 \in Location &: \texttt{disjunct}(l_1, l_2) = true \\
&\Leftrightarrow (\forall l_3 \in Location : \\
&\neg((\texttt{contains}(l_1, l_3) = true \vee l_1 = l_3) \wedge \\
&(\texttt{contains}(l_2, l_3) = true \vee l_2 = l_3))) && (4.9)
\end{aligned}$$

For convenience we can introduce an additional operation `overlaps` which evaluates to $true$ whenever `disjunct` evaluates to $false$ and vice versa.

Analogous to geometrical models, as defined in the previous section, we can define an additional operator `myLoc`.

DEFINITION 4.3.7 **Location Reference.**

$$\texttt{myLoc} : ObjectId \rightarrow Location$$

Given the above definitions, now it is possible to define the notion of a location graph as a formal representation of a specific set of locations and their relationships to each other. We use this definition later in this chapter, as well as in Chapter 6.

DEFINITION 4.3.8 **Location Graph**

A Location Graph is a graph structure $Loc = (L, C)$. *Whereby* L *is the set of possible locations as defined above and* C *the set of* edges *between any two locations* l_1, l_2 *from* L, *where* $\texttt{contains}(l_1, l_2) = true$.

4.4 A Reference Location Model for Publish/Subscribe Middleware

In this section we present the reference location model for the use in the publish/subscribe middleware. The location model of choice must facilitate two main properties:

- *Flexibility.* By definition, middleware has to be flexible and extensible. This is especially true for a publish/subscribe middleware for dynamic environments. When considering the choice of a data model for representing location information, it is important to use a model that is as versatile as possible without hindering the operation of the notification service. Especially in pervasive environments, we have to take into account that (pairwise) consumers and producers of data are not necessarily part of the same application domain.

- *Efficiency.* The item above states that the data model should be flexible and extensible. On the other hand, the trade-off at this point is the efficiency of the filter model operating on location data. For efficient routing it is mandatory to have a combination of data model and filter model complementing each other. Therefore, the overhead necessary to evaluate a location-dependent filter should be minimized where possible.

Effectively, the ideal location model adheres to the above properties and is convenient to use for application-specific behavior at the same time. Obviously, this is the proverbial "quest for the holy grail". On the application level, we *might* want to specify location on a semantic level and in the infrastructure, we need location specifications to operate on the same syntactic level for comparability. However, in the previous Sections 4.3.3 and 4.3.4 we have introduced two models for the specification of location. The geometric model of Section 4.3.3 is operating on a flat and unstructured data model which can be evaluated efficiently, given easy-enough geometrical shapes to compare. On the other hand, the symbolic location domain model, as presented in Section 4.3.3, exhibits a strong structure and semantic meaning to a certain degree. This makes it suitable for the specification of location for applications.

4.4.1 Interoperability of Location Models

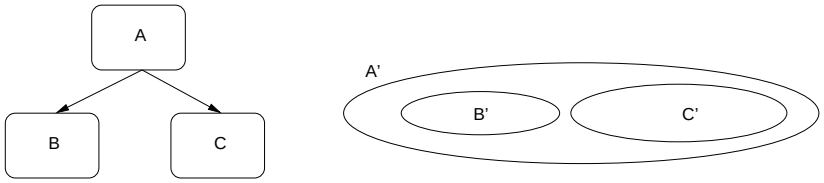
The best choice as a reference location model for the use within the core of the distributed notification service is the geometric model of coordinate tuples and geometric shapes and the location domain model as (optional) specification location model for the use in applications.

The challenge we face at this point is to show that *bijective* translation operations between these two models can be defined. Luckily, as the intended similarities in the specifications of the location models in Definition 4.3.2 and Definition 4.3.6 indicate, bijective mappings can be found (cf. also [Leo98]).

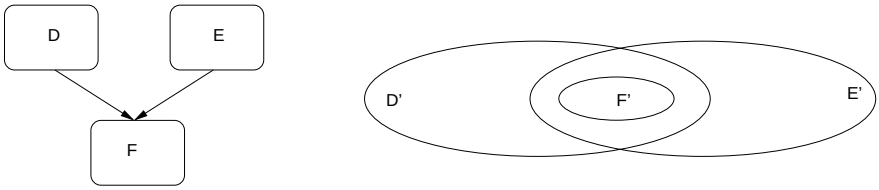
To show that both models can be mapped onto each other we have to recall the underlying property of both models: spatial extension in physical space. Thus, in both models, Area and Location, respectively, reference a well-defined partition of space. In Figure 4.4 we have illustrated the basic idea of the mappings.

Specification. First, we introduce a mapping from the location domain model into the geometric model. Please note that in the following we label the operations `contains`, `overlaps`, and `myLoc` according to their originating location model.

DEFINITION 4.4.1 **loc2Area.**
Given a Location Graph Loc*, the mapping* `loc2Area` *has the following properties:*



(a) Containment and Disjunction



(b) Overlapping

Figure 4.4: Mappings of location models

$$\texttt{loc2Area} \quad : \quad Location \rightarrow Area$$

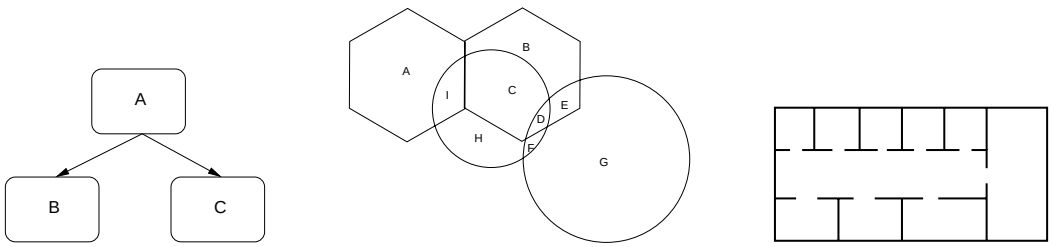
$$\begin{aligned} \forall l_1, l_2 \in Location \quad : \quad & \texttt{contains}_{sym}(l_1, l_2) = true \\ & \Rightarrow \texttt{contains}_{geom}(\\ & \texttt{loc2Area}(l_1), \\ & \texttt{loc2Area}(l_2)) = true \end{aligned} \tag{4.10}$$

$$\texttt{loc2Area}(\texttt{anyLoc}_{sym}) \quad = \quad \texttt{anyLoc}_{geom} \tag{4.11}$$

`loc2Area`'s semantics is that given a location graph Loc we can translate the disjunct locations contained in Loc into the respective areas as defined in the geometric model. It should be noted that the location graph Loc can be assumed to be static and therefore the mapping `loc2Area` can be computed in advance.

A similar mapping can be found for the opposite "direction." The challenge here is to find a well-defined semantics if the area specified does not exactly match a location as specified in the location graph. Then, the location graph has to be searched for the least location that covers the area specified. As we assume to have a well-defined root-location in the location graph, i.e., `anyLoc`, such a location always can be determined.

DEFINITION 4.4.2 **area2Loc.**
Given a Location Graph Loc, the mapping `area2Loc` *has the following properties:*



(a) Containment graph (b) Zone graph (c) Floorplan

Figure 4.5: Location Graphs

$$\texttt{area2Loc} \quad : \quad Area \rightarrow Location$$

$$\begin{aligned} \forall a \in Area, l \in Location \quad : \quad & \texttt{area2Loc}(a) = l \\ & \Leftrightarrow \texttt{contains}_{geom}(\texttt{loc2Area}(l), a) = true \wedge \\ & (\forall l' \in Location : \\ & \texttt{contains}_{geom}(\texttt{loc2Area}(l'), a) \\ & \Rightarrow (\texttt{contains}_{sym}(l', l) \vee l' = l)) \qquad (4.12) \end{aligned}$$

$$\texttt{area2Loc}(\texttt{anyLoc}_{geom}) \quad = \quad \texttt{anyLoc}_{sym} \qquad (4.13)$$

The definition above states, that the set of possible covering locations for area a must be searched until a location that covers a is found. Additionally, the mapping searches for the optimal matching location. For efficiency reasons this requirement might be omitted. The case of translation from the geometric model into the symbolic model usually occurs on the borders of the event broker network, where application-specific location models are in use. Hence, for efficiency, we can piggyback the original location description and replace the result of the geometric mapping again with the original value, when a notification is delivered to a client. Thus, we do not necessarily need an additional mapping operation.

With Definition 4.4.1 and 4.4.2 we have introduced means to map different data- and semantic models of location onto each other. Additionally, the definitions given are suitable to serve as a basis for the implementation within the filter model of the notification service.

4.4.2 Location Graphs

In this subsection we introduce *location graphs*, as a means to facilitate location-aware notification delivery as introduced in Chapter 6. In Definition 4.3.8 on Page 64 we have introduced location graphs for symbolic location models. There, the location graph is comprised of the symbolic locations and the containment relationships between them (cf. Fig 4.5(a)). Another example for a

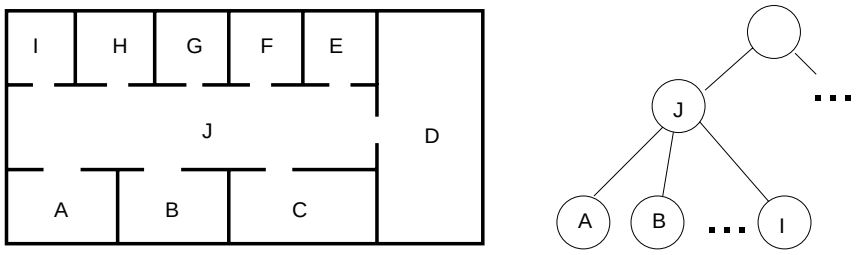


Figure 4.6: Floor plan and associated location graph

location graph is described in Section 4.3.4.2 and shown in Fig 4.5(c) where we have seen that *zone models* constitute a natural location graph due to their discrete structuring of space and location.

However, for the use within the publish/subscribe notification service we need to provide a graph structure which serves two purposes:

- Complementing the chosen location model in the infrastructure, i.e., the provision of information where information is produced and consumed, and
- the description of possible and probable changes of location of entities over time.

Ideally, a location graph provides enough information to facilitate support for *mobility*, i.e., change of locations. Thus, the location graph is a partition of potentially continuous movements through space into discrete locations. Abstractly, a location graph constitutes a *state chart* with respect to location changes according to a certain location model and its metric.

Obviously, we have to distinguish between application specific location graphs and system specific location graphs. An example of the former is a location graph as it is produced by a route planning application. Usually, in such an application only direction changes are indicated, like, changing to a different highway. In this respect, a change of direction constitutes a location, rather than the measurement in miles (reactive behavior). On the other hand, given a fixed location, like a building, the notification service and its brokers are associated to a certain model of location to optimize notification delivery (cf. Figure 5.8 on Page 95). We have seen before that in symbolic location models the graph structure typically follows the concrete layout of the physical space it models. The same idea can be applied to the layout of the broker network to draw from message delivery localities for the support of mobile clients, as we will show in Chapter 5 and 6.

Thus, at the junction between an application-specific location graph and the system-wide location graph, we need to specify the trade-off between those two notions of location graphs.

Before going into details we introduce *granularity*, describing the smallest "unit" for modeling location.

DEFINITION 4.4.3 **Granularity and minimal matching filters**
For a given location graph Loc the Granularity $\mathcal{G}(Loc)$ *denotes the set of smallest possible coverings of space for the location model used within the notification service, i.e., the* minimal matching filters *for a location graph Loc.*

A *minimal matching filter* is a reification of a node of a location graph within the filter model used in the publish/subscribe infrastructure.

For a given location graph $Loc = (L, C)$ that is associated with a space, a smallest matching filter for a location $loc \in L$, i.e., a node in Loc, matches loc and all contained locations within, that are not modeled as node in the location graph.

For example in Figure 4.6 on Page 68, the floor plan on the left-hand side can be translated into the location graph $Loc_{floorplan}$ as shown on the right-hand side. Thus, $\mathcal{G}(Loc_{floorplan}) = \{A, \ldots, J\}$, whereby $\{A, \ldots, J\}$ constitute the minimal matching filters for $Loc_{floorplan}$.

Consequently, we allow to define a filter matching only, e.g., location A, but not to define a finer-grained filter for a table contained in A. The only way to allow for this is either (i) to subdivide the system-wide location graph until the granularity $\mathcal{G}$ also matches the table; Or (ii) to require applications to be able to do client-side filtering and filter out unwanted notifications. "Table" as a discernable location, obviously is only required for an application-specific location model. Otherwise, (i) would have been applied.

4.4.3 Location-Dependent Filters

Besides the specification of location and restricting the flow of notifications to a certain partition of space, the location graph also serves the purpose of allowing for the special marker `myLoc`, as introduced above in Definition 4.3.3 on Page 61 and Definition 4.3.7 on Page 64.

We allow clients to specify a placeholder within a subscription, referencing their current location. From the viewpoint of an application this constitutes a way to be *location-aware* while moving around, e.g., in a house or office space.

A publish/subscribe system that offers location-dependent filters has the same interface as a regular publish/subscribe system (i.e., it offers the *publish*, *subscribe*, *unsubscribe*, *notify* primitives). However, in specifying subscription filters for name/value pairs referring to "*location*" it supports a new primitive to specify things like "all notifications where the attribute *location* equals my current location". More precisely, we postulate a specific marker *myLoc* that can be used in a subscription. The marker stands for a specific set of locations that depend on the current location of the client. Again in the "Free Parking Spaces" service, a client could issue a subscription for all free parking spaces in the vicinity of his current location as follows: (*service* $=$ "parking"), (*car-type* $\geq$ "compact") for the application domain-specific part of the service and (abstractly) (*location* $\in$ `myLoc`) in the envelope.

The set of locations associated with the marker is taken from a particular range L of locations for a given location graph Loc. This set can contain all the different rooms of a building, all the streets of a town, or all the geographical coordinates given by a GPS system up to a certain granularity.

Given a notification with the attribute *location*, the subscription (*location* $\in$ `myLoc`) will evaluate to $true$ for a particular client C at location y if and only if $C \in$ `myLoc` where `myLoc` is the specific set of locations associated with a client C. In this case we say that the notification matches the location-dependent filter.

In the car example, the car driver looking for a parking space might want to specify:

(*location* $=$ "at most 0.5 miles away from *myLoc*")

In this case, *myLoc* corresponds to all elements of L that satisfy this requirement.

A tentative but incomplete solution for location-dependent filters. While location-dependent filters are not directly supported by current publish/subscribe middleware, one might argue that it is not very difficult to emulate them on top of currently available systems in this case.

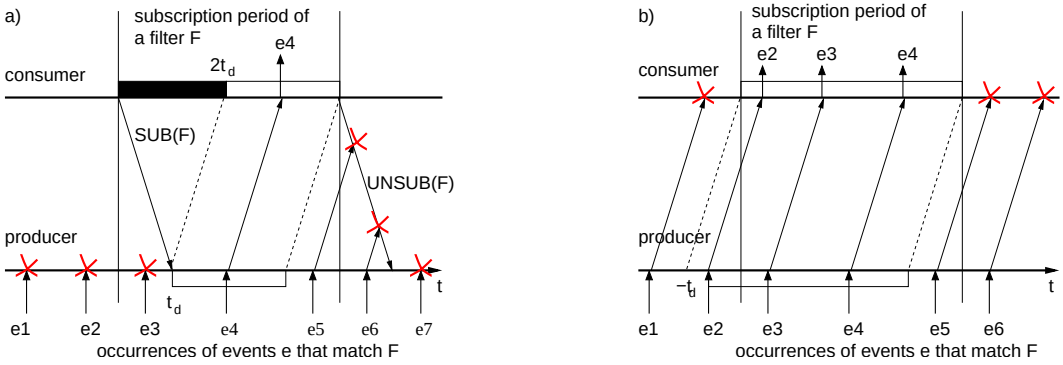


Figure 4.7: Blackout period after subscribing with simple routing a) and flooding with client-side filtering b).

The idea would be to build a wrapper around an existing system that follows the location changes of the users and transparently unsubscribes to the old location and subscribes to the new one when the user moves. However, depending on the internal routing strategy of the event system, it may lead to unexpected results. The routing strategies deployed in many existing content-based event systems such as Siena [CRW01], Elvin [SA97], and REBECA [FM00] lead to blackout periods where no notifications are delivered. The problem is that it usually takes an non-negligible time delay to process a new subscription. After subscribing to a filter, it takes some time t_d until the subscription is propagated to a potential source. Then it takes at least another t_d time until a notification reaches the subscriber. This phenomenon is depicted in Figure 4.7a. (Note that the delay t_d may be different for different notification sources and may change over time.) If the client remains at any new location less than $2t_d$ time, then the subscriber will "starve", i.e., it will receive little or no notifications. However, in the context of location tracking in Section 4.4.4.2 we have specified exactly this solution to clarify the semantics of the `myLoc` marker in conjunction with location services and location tracking services. An algorithm for the solution described here is given in the Figures 4.10 and 4.11 on Page 76 and Page 77, respectively. Later in this thesis (cf. Chapter 6), we will revisit this solution and extend the solution proposed in the Figures 4.10 and 4.11 to avoid the shortcomings of this tentative solution.

An intuitive but inefficient solution. Another basic solution that can be immediately built using existing technology is based on flooding notifications through the complete network. The local broker can then decide to deliver a notification to a client depending on the client's current location (see Figure 4.7b). Obviously, flooding prevents the blackout periods, which were present in the previous solution, but it should be equally clear that flooding is a very expensive routing strategy especially for large publish/subscribe systems [MFGB02].

On quality of service. Interestingly, while flooding is very expensive and therefore not desirable, it comes very close to the quality of service that we would like to achieve for mobility together with location-dependent subscriptions, namely to the notion of being subscribed to information "everywhere, all the time". The problem is that it is hard to precisely define the behavior of flooding

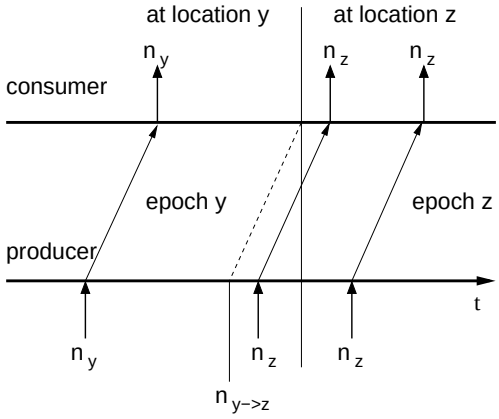


Figure 4.8: Defining the quality of service for logical mobility using virtual notifications $n_{y \to z}$ that arrives at the consumer just at the time of the location change from y to z.

without reverting to some unpleasantly theoretical constructions of operational semantics.

The quality of service we require for location-dependent subscriptions therefore is simply stated as follows: On change of location from x to y, all notifications should be delivered to the consumer "as if" flooding were used as underlying routing strategy. This statement is made a little more concrete in Figure 4.8 where the sequence of notifications generated by any producer is divided into epochs that correspond to when the notification actually arrives at the consumer (the epoch borders between location y and z are drawn as a virtual notification $n_{y \to z}$). We require that all notifications matching the current location-dependent subscription from every such epoch must be delivered. Intuitively, the epochs define the semantics of flooding.

4.4.4 Location Specification

On the basis of the bijective mappings introduced above in Section 4.4.1, we are able to allow the specification of location-dependent filters and notifications in terms of (i) the geometric model, as well as (ii) the location domain model.

However, for both cases we need a specification language (or notation) for location-dependent information. In the literature on location services and here again in [Leo98], specification languages are detailed that take into account certain properties of location, uncertainty, and location models. Therefore, for simplicity, in this thesis we follow this specifications and adapt them where needed. As a basis serves the specification as introduced by Leonhardt [Leo98] but was extended for proper use together with a notification service.

4.4.4.1 Naming

We want to enable applications (and users) to use a rich namespace to name locations. On the other hand, due to the need for bijective mappings and thus the limitation to the location domain model, we have to restrict the structure of the namespace sufficiently to maintain consistency.

However, user-friendly naming in the general case of context-aware environments was extensively examined and prototyped in [Huh01]. Based on [Leo98] and [Huh01], in this thesis, we restrict the general notion of naming to the special case of naming of locations.

For the specification of a location, we need an area and a position[3]. The area can be specified as geometric area as defined above. A position can either be another geometrical specification or a location concept. Thus we define:

DEFINITION 4.4.4 **Location notation:**

$$
\begin{aligned}
\text{location} &::= \ < \text{area} > \ @ \ < \text{position} > \\
\text{area} &::= \ < \text{location concept} > \ | \ < \text{geometric definition} > \\
\text{position} &::= \ < \text{located object} > \ | \ < \text{fixed position} > \\
&\quad | \ \texttt{myLoc} \\
\text{fixed position} &::= \ < \text{geometric position} > \ | \ < \text{symbolic position} > \\
\text{geometric position} &::= \ < \text{reference system} > ' : ' < \text{coordinates} > \\
\text{symbolic position} &::= \ (< \text{symbolic position} > ' \ / ' < \text{label} >) \\
&\quad | \ < \text{well known position} >
\end{aligned}
$$

The definition above enables us to specify location predicates in filters. For example, we can specify:

$$< 5\text{m@university/elsewhere/wilhelminenstr/floor } 1/\text{room } 120$$

to define a radius of 5m around the room C120 (the author's office).

Another, more interesting example is:

$$< 500\text{m@}\texttt{myLoc}$$

Used in a location-dependent filter the above location predicate constrains matching to notifications originating from a radius of 500 meters around the current position.

4.4.4.2 Location Tracking

The example of using `myLoc` within a location-dependent subscription poses the question of how such a position marker actually is kept up-to-date.

Due to the inherent generality of the notification service specified in this thesis and the use of mixed location models, we have to apply a divide-and-conquer strategy. An instance of a notification service should not be tied to a particular location tracking or positioning system. Otherwise such a tie would be contradictory to the nature of the notification service as an open and extensible platform. The problem is that a location tracking system might evolve over time or is exchanged by another system. In such case, any close ties to a particular tracking system would render the notification service useless for location-sensitivity.

However, for location-dependent subscriptions, we rely on the capability to track an entity's location and update the `myLoc` marker accordingly. Fortunately, we defined `myLoc` in a way that it

[3] Relative to a reference positioning system.

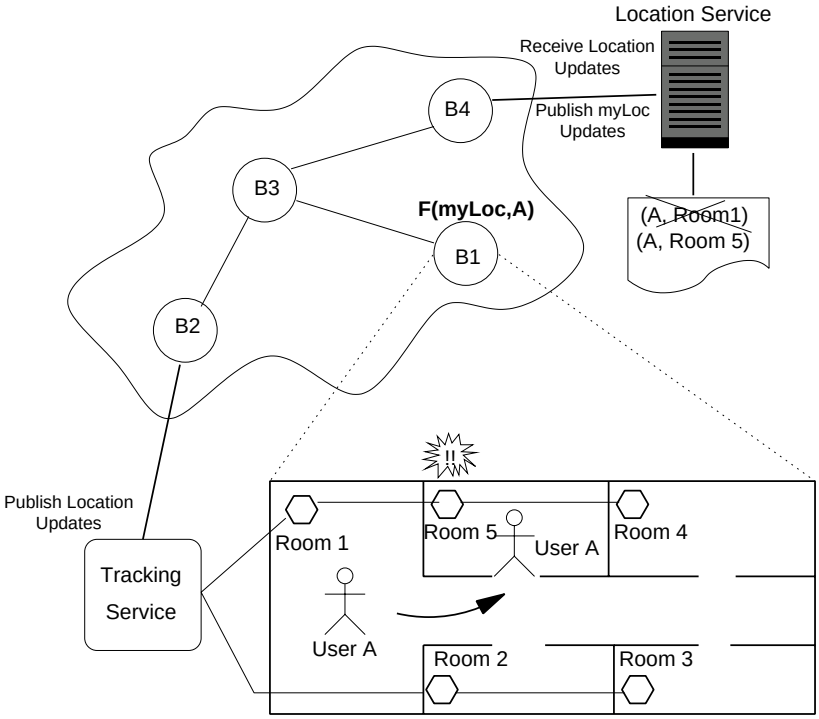


Figure 4.9: Location detection and `myLoc`

is a mapping from an $ObjectId$ to an area or location, respectively. This enables us to specify an interface to an external location tracking system, such as the location service specified in [Leo98]. Additionally, the location model introduced in this chapter basically is using comparable abstractions. Hence, delegation of location tracking to an external location service is not only feasible, but also desirable to cleanly separate concerns. Thereby, we also promote extensibility and openness.

For the sake of simplicity, we do not consider issues of *Universally Unique Object Identifiers* (UUID) and require the object identifier used in a location-dependent filter to be the same as it is used within the location service. Given this simplification, we can define the semantics of `myLoc` appropriately.

Consider the situation as shown in Figure 4.9. Client A has issued a location-dependent subscription $F(\texttt{myLoc}, A)$ for some location-related information. Let us further assume that Broker $B1$ is the broker allocated for the location graph shown (cf. also Chapter 6 for more details). $B1$ therefore is responsible for adapting `myLoc` to the current position of client A.

Additionally, we have shown two more infrastructural components: a *location tracking service* and a *location service*, respectively. Both are specialized components to provide in their combination a reliable source for client position sightings and updates. A detailed specification of their functionality is out of scope of this thesis. For a survey of basic location tracking technologies refer to, e.g., [HB01b; HB01a; Leo98].

Nonetheless, we can assume that both rely on the specified notification service as underlying

communication infrastructure and therefore are aware of its basic properties. Please note that neither the location tracking system, nor the location service have to be attached to the same broker as client A. They constitute orthogonal concerns and therefore are not subject to any location model or comparable metric. We have indicated this by assigning different brokers to both ($B2$ and $B4$).

Given the setting as described above, we assume that client A is changing location from "Room 1" to "Room 5". The location system, e.g., an *Active Bat* [WJH97; ACH+01] or *Cricket* [PCB00] system, detects the change of location and reports the new position of client A to the location tracking service. The location tracking service generates an *application domain-specific* location update event and publishes N with $\texttt{publish}(N(A, \text{'Room5'}))$ as notification into the publish/subscribe infrastructure. The notification service now dispatches the notification N as standard non-location related information, as the knowledge about the location is part of the application domain, shared between location tracking service and location service[4].

Additionally, the *location service* is by default subscribed to any location update events generated by the tracking service in order to keep the locations of clients up-to-date. Hence, the sent notification N eventually is received by the location service. The information contained in the notification N is used to update the database of $(client, location)$-tuples. Moreover, the location service itself publishes an additional event N', containing the same information as N, but restructured as special `myLoc` inter-broker message, where the location information is made explicit in the notification's envelope. This information N' is published into the notification service and used to update the `myLoc` markers where appropriate. The general data structure of N' with respect to the `myLoc` marker is $N'(< ObjectId >, < location >)$, with $ObjectID$ and $location$ as introduced above. Details on the implementation can be found in Chapter 9.

Every broker hosting clients that have issued location-dependent subscriptions subscribes to these inter-broker messages with the $ObjectId$ of their respective clients, thereby establishing a routing path for location updates from the location service to the broker hosting the client. This last step establishes the intended semantics of `myLoc`.

The algorithm every broker has to implement to facilitate location updates is summarized in the Figures 4.10 on Page 76 and 4.11 on Page 77. A problem that arises when adhering to the asynchronous publish/subscribe paradigm is the initialization of a new `myLoc` marker. For now, we assume a *heartbeat* mechanism, which regularly sends the current position of clients into the network. Some triggered update mechanism is necessary as otherwise a currently non-mobile client would never cause a *location update* notification. Then the current location would not be resolvable at all. However, in Chapter 9 we introduce an implementation that uses a listener design pattern approach to resolve this issue more elegantly.

Movement Graphs. In Chapter 6 we use *movement graphs*, i.e., a complementary graph structure to the location graph, to optimize responsiveness and setup times for moving clients using the `myLoc` marker. While the location graph basically describes the physical and logical containmentships of locations, the movement graph describes the neighborhood and reachability of locations. Obviously, a movement graph can easily be derived from an available location graph. Once a movement graph is instantiated it can be put to use for movement prediction and message delivery optimization as we will show later in this thesis.

[4] It is worthy to note that interaction between the two services also might be peer-to-peer. The use of the notification service is for demonstration purposes only.

4.5 Summary

In this chapter we have laid the foundations of context and location-dependent message delivery. We have analyzed the concept of context-awareness from the viewpoint of a publish/subscribe infrastructure. The key categories of context we identified are *application domain-specific* and *system-specific context*. The main distinction we elaborated on is the usability of a certain context information as an *index* or *addressing criteria* for message routing. A context-aware publish/subscribe infrastructure tries to optimize routing of messages by drawing on context information.

After identifying the key issues of general context-awareness in the light of the event broker network, we narrowed the view to location-awareness as the prime source of context-awareness. We emphasized the outstanding importance of location as a convenient index on context information.

Our discussion motivated the choice to build support for location-awareness into the core of the event broker network by making location a first-class concept directly usable in subscriptions and notifications. However, to do so, first we had to understand the implications of this approach by analyzing the design space of location in the face of the asynchronous publish/subscribe paradigm. In our taxonomy of location models we have shown that the two basic models usable are *geometric models* or *symbolic models* of space and location.

We introduced both models in greater detail and analyzed their strengths and weaknesses to assess their potential as *reference location model* for location-aware notification delivery. The analysis showed that a geometric model of space and location is the natural choice for the use within the routing network; but on the other hand, a symbolic model is the model of choice for the use in applications. Thus, we settled on a hybrid model, the *location domain model*. It satisfies both requirements by adding enough constraints to a symbolic model to provide bijective mappings between the pure geometric model we use in the infrastructure and a more intuitive symbolic model used on the application level.

Finally, we outlined the technical implementation by introducing the notion of location-dependent filters and location-/movement graphs as a means to structure the layout of a distributed event broker network according to the location model and movement restrictions faced in the “real” world (cf. also Chapter 5 and 6). Furthermore, we showed that the graph structures can be exploited to model a special new functionality in the broker network: the `myLoc` marker. This marker is a placeholder used within a location-dependent subscription. The semantics is an automatically updated location reference. With such a reference to the current location, location-awareness is feasible. To emphasize the need for support in the infrastructure, we showed where the “naïve” solutions to implement such a behavior fall short. Additionally, we gave an algorithm for the actual implementation of `myLoc` in the brokers of the notification service. We will revisit and eventually extend the approach of `myLoc` in Chapter 6.

```
/** upon receiving myLoc subscription (C, F(myLoc))
 * via link L_N */
void receiveSub(C, F(myLoc), L_N) {
  routeTable.add(C, F(myLoc), L_N); // add new subscription
  propagate(C, F(myLoc), L_N); // to all neighbor brokers with
                               // matching advertisements except L_N
  if (!localClient.contains(C) {
     createSubscription(this, locationUpdate(C)); //
                               // listen to all location update events for a given client C
     createSubscription(this, locationHeartbeat(C)); //
                               // listen to all heartbeat events for a given client C
     routeTable.setInit(C, F(myLoc), false); // set the
                               // Filter to inactive until current location is initialized
     routeTable.setActive(C, false); // Set the client's status
  } else { // client is known; administration done
     routeTable.setInit(C, F(myLoc),routeTable.getActive(C)});
              // Initialize filter according to client status in broker

  }
}

/** upon receiving notification n(loc) from B_j */
void receiveNotif(n(loc),B_j) {
  routeTable.route(n(loc)); // inspect all entries with myLoc
                                        // and route accordingly (see below)
}

/** upon receiving myLoc update message */
void receiveUpdate(n) {
  routeTable.updateLocation(n.objectId, n.location); // update all entries
                               // where the client is n.objectId
                               // to the new location
}

/** upon receiving heart beat message for client ObjectId */
void receiveHeartbeat(n) {
  routeTable.updateLocation(n.objectId, n.location); // update all entries
}

/** upon timeout of C */
void receiveTimeOut(C){
   routeTable.remove(C, *);

}
```

Figure 4.10: Basic algorithm for myLoc subscriptions.

```
/** routing with location attributes */
class RouteTable {

  ... // ommmited details

  void route(Notification n) {
    if (n is n(loc)) { // has location attribute
      for (∀f ∈ filters with myLoc assigned) // (myLoc) subscriptions
          if (contains(f.myLoc.current(), n.location()))
            // is the notification contained by the current location of the client?
            if (f.match(n.content())) // then deliver iff content matches
              deliver(f, n);
    } else {
      // normal operation w/o location
    }
  } // end of route()

  void updateLocation(ObjectId id, Area loc) {
    for (∀f ∈ filters of client ObjectId &&
          myLoc assigned) { // all location-dependent filters for client C
        if (!f.myLoc.initialized()) // not initialized
            f.myLoc.initialized := true; // set to initialized and active
        f.myLoc.current := loc; // assign the new location
        propagate(C,f); // propagate change of location
    }
  } // end of updateLocation()

} // end of class RouteTable
```

Figure 4.11: Extension to the class RoutingTable

5 Mobility Extension for a Distributed Notification Service in Mobile Environments

You know what a "legacy application" is?
It's one that works.

Bill Cafiero, co-chair of many ANSI ASC X12 task groups

5.1 Introduction

In this section we focus on extending the REBECA model of a distributed notification service to support *mobile clients*, i.e., mobile consumers and producers of data which are used in mobile settings. However, up to now research in event systems has mainly focused on using publish/subscribe middleware in rather static, non-mobile environments, i.e., systems where clients do not roam and the infrastructure itself stays rather fixed or is only changing slowly during the system's lifetime. Consequently, most publish/subscribe infrastructures (cf. Chapter 3) have optimized algorithms for information delivery in those settings. Support and optimizations for mobile clients are no built-in features of the infrastructure; it is left to the applications to adapt or reissue subscriptions.

Obviously, the first step towards mobility is to make the publish/subscribe middleware mobility-aware. Therefore, we need to add support for roaming clients and their needs. From the application's point of view, making use of the inherent loose coupling, we should be able to continue to use existing, successfully deployed applications that are based on the publish/subscribe paradigm without having to rewrite them ("legacy" applications). A very simple example is the famous stock quote application, seamlessly transferred from a non-mobile desktop PC to a mobile device, such as a PDA (cf. Fig. 5.1).

As a first step and basis for future developments, applications should not need to be aware of mobility, carrying on their processing independent of their location. On the other hand, some applications rely on the awareness of mobility and their sourrounding, e.g., context-aware applications. Those applications have different needs than the applications assumed in this chapter and will be the central topic of the Chapters 6 and 7. Another aspect is that not every subscription issued by a context-sensitive application is necessarily related to the current context; such applications might want to hide some aspects of mobility in the infrastructure, in cases where awareness of mobility and the current surrounding is not exploited.

In this chapter we introduce mechanisms, built on top of a notification service, especially designed to take care of subscriptions issued by mobile devices. Subscriptions and related notifications are relocated from broker to broker, while users and devices roam the physical world connecting to different brokers along their itinerary, effectively achieving complete transparency of mobility.

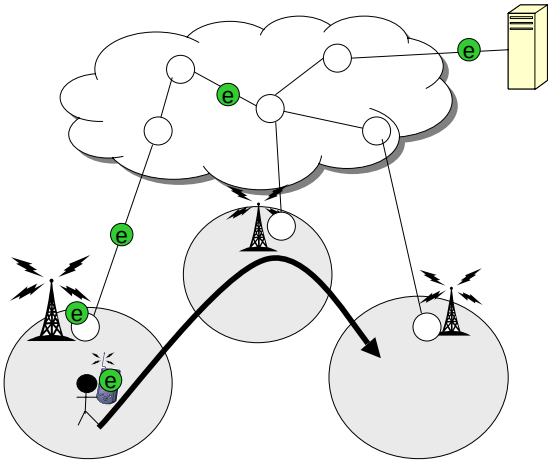


Figure 5.1: Location-dependent notification delivery in physical mobility

For the provision of location transparency the distributed notification service has to arrange for handling the unavoidable side-effects inherent to mobile environments. They must not have to be the concern of the application layer. Side-effects include phenomena like low bandwith lines or transient phases of disconnected operation. Thus, to meet the requirement of location transparency, the notification service has to be adapted for this purpose by introducing special means of mobility support into the network of brokers. The foremost need we identified is transparent connection handling or, to be more precise, handling the loss and re-establishment of connections. Another concern for the infrastructure is the handover of subscriptions from one to the next border broker that is currently hosting the client, when an application on a device is mobile. Therefore, we need mechanisms for rerouting and replaying of notifications to different locations in the broker network.

The general structure of this chapter is as follows: the next section analyzes the requirements for mobility support in more detail. However, the heart of this chapter is Section 5.3. There, we show how the brokers constituting the distributed notification service have to be extended to cater for mobile clients. After giving more details in subsequent sections, in Section 5.5 we discuss how our extensions meet the requirements stated before and argue that it optimally facilitates transparent mobility support for mobile clients. Finally, we compare the solution presented in this chapter to alternative approaches and discuss where they fall short in comparison to our approach. Thereby we clearly state our contribution to the field of mobile publish/subscribe systems.

5.2 Analyzing the Requirements

The general requirements introduced in the previous section share some similarities to what in the area of mobile computing is called *terminal mobility* or *roaming*. A client accesses the system through a certain number of *access points* (GSM base stations, WLAN access points, or in our case, border brokers). When moving physically, the client may get out of reach of one access point and move into the reach of a second access point with possibly non-overlapping reaches. In the remainder

we will address this with the term *physical mobility* or *roaming*. In general we cannot expect to have seamless access to the broker network but more a sequence of phases of connectedness, e.g., on the daily route between home and office. In this setting we analyze the quality of service requirements from the viewpoint of roaming clients.

Transparency of Mobility. One of the prime advantages of transparent mobility support is the usefulness for legacy applications. However, the publish/subscribe paradigm is well established in *non-mobile* setting, hence, in general legacy applications are not aware of mobility. If adding mobility support should require a change of interface between applications and publish/subscribe infrastructure, without redesign of applications a changed interface would render legacy applications useless. Obviously, this requirement is not necessary for mobility-aware applications that want to deligate some aspects of mobility. We state this result in Requirement 5.2.1:

REQUIREMENT 5.2.1 **Interface.** *The interface for applications to the publish/subscribe system must not change.*

Completeness. Despite intermittent disconnects, the publish/subscribe middleware should deliver all notifications for a client eventually. This is the core requirement for achieving *transparency* of mobility. Usually, in static, non-mobile scenarios a client can assume reliable delivery of notifications. Unfortunately, reliability is a scarce resource in mobile settings. However, for maintaining transparency of mobility, the *impression* for applications of reliable message delivery must be built on top of the notification service in the mobile case. This is summarized in Requirement 5.2.2

REQUIREMENT 5.2.2 **Completeness.** *Despite intermittent disconnects, the publish/subscribe middleware delivers all notifications for a client eventually.*

Ordering In Section 3.3 sender FIFO ordering was guaranteed for the classical non-static case. Requirement 5.2.3 below introduces the same guarentee as an eligible feature in the mobile case, too. In the static case no "session management" was necessary to rely on correct order delivery of notifications originating from a single source[1].

REQUIREMENT 5.2.3 **Ordering.** *Sender FIFO ordering as in the non-mobile case remains intact.*

Responsiveness The delay of relocating a roaming client should be minimal to maximize the responsiveness of the system. This has to be taken into account when designing a relocation protocol. Usually, due to the distribution and inherent overhead of mobility support, especially the latency of notification delivery can be expected to be higher. On the other hand, the basic REBECA model of notification delivery explicitly does not assume an upper bound on message delivery over any link in the network (cf. Section 3.3). Hence, any application using the standard REBECA notification service has to be designed taking deferred delivery into account.

REQUIREMENT 5.2.4 **Responsiveness.** *The delay of relocating a roaming client should be minimal to maximize the responsiveness of the system.*

[1] As already mentioned, this does not mean that a global ordering can be achieved in case of multiple sources.

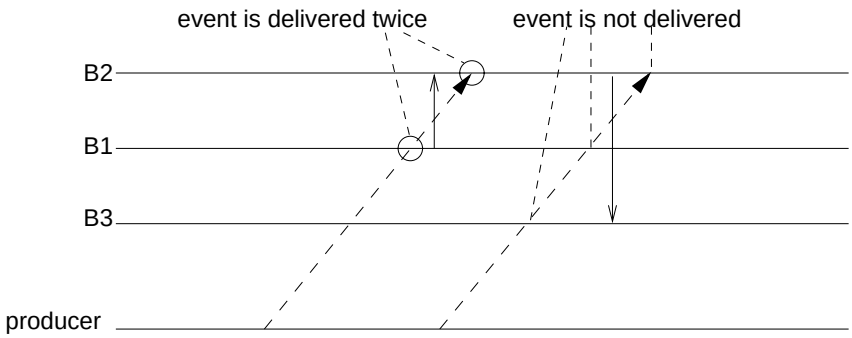


Figure 5.2: Missing notifications in a flooding scenario.

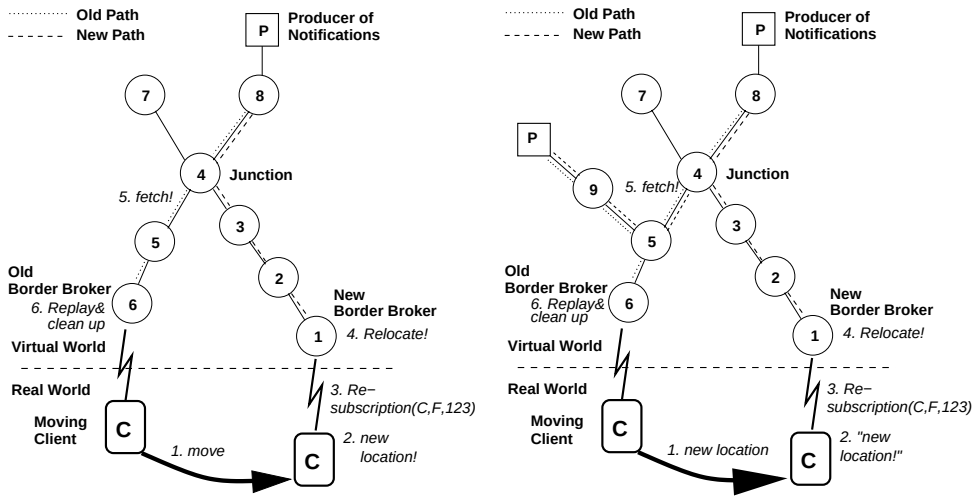
Informal invariant derived from the requirements. To maintain a connection and to hide phases of disconnectedness, the responsibility of providing access to the system is usually handed over from one access point to the other (involving some handover protocol). Overall, the invariant of the algorithm presented throughout the next sections is that any application should have the *impression* to a reasonable degree that it is operating in a rather reliable, static, non-mobile environment, where erroneous states (e.g., notification loss) might happen but should not happen regularly.

Possible naïve solutions. One solution would be to rely on Mobile IP [Joh95] for connecting clients to border brokers, hiding physical mobility in the network layer. The drawback, however, is that the communication is also hidden from the publish/subscribe middleware, which is then not able to draw from any notification delivery localities or routing optimizations, thereby possibly violating the requirement of responsiveness. Such an approach might only be feasible if the physical and logical layout of a given system is completely orthogonal.

A different, naïve solution to implement physical mobility would be to use sequences of *sub-unsub-sub* calls to register a client at a new broker. When a client moves from border broker B_1 to B_2, it simply unsubscribes at B_1 and (re-) subscribes at B_2, without any support in the middleware. But a client may not detect leaving the range of a broker and is in this case not able to unsubscribe at its old location. Even more severely, during its time of disconnectedness, the client might miss several notifications or get duplicates, even if notifications are flooded in the network and the location change is instantaneous. This problem is depicted in Figure 5.2. Hence, this solution is not complete and we outline an algorithm in Section 5.3 that takes into account all requirements stated above.

5.3 Notification Delivery with Roaming Clients

In this section we introduce an algorithm for extending standard REBECA brokers to cope with mobile clients, maintaining their subscriptions as well as guaranteeing the required quality of service that was described in the previous section. Apart from guaranteeing uninterrupted notification delivery, our algorithm also ensures that the old border broker will eventually receive an equivalent to an explicit *sign-off* from the client even if an explicit unsubscribe was not possible.



(a) Single Producer (b) Multiple Producer

Figure 5.3: Moving client scenarios with one and multiple producers

By design, the mechanism we use introduces a natural way of distributed caching, which seems in general preferable to a potentially problematic central caching proxy. Moreover, our solution even supports a correct handling of a multiple-producer scenario as shown in Fig. 5.3(b).

5.3.1 Algrithm Overview

The routing network of REBECA was extended to implement an algorithm consisting of three distinct phases, *propagation*, *fetch*, and *relocation*. Using exclusively the publish/subscribe paradigm together with the distributed broker network, each phase has a separate goal.

- *Propagation.* The goal of the propagation phase is basically twofold. In Figure 5.3(a) one can see that, after a client is reconnecting to a different broker, a new path to one or more producers of requested data must be set up. However, due to the special structure of the broker network this path is meeting the old delivery path at some point. We call this particular broker the *junction broker.* By identifying the *junction* where old and new path meet, the propagation phase is finished and a new delivery path is set up.
- *Fetch.* After the identification of the junction a special *fetch* message is sent along the old delivery path, with the goal of shutting down the old delivery path and, more importantly, identifying which part of the old delivery path can be discarded and which part has to be redirected. This is the case in a multiple producer example as shown in Figure 5.3(b). After the fetch message reaches the border broker of a relocating client C, the second phase terminates.
- *Relocation.* The last phase is the actual relocation of cached messages for client C. A standard replay message as already being part of REBECA is used to sent messages from the old location

to the new location. An additional goal is to "garbage-collect" those parts of the old delivery path between junction and border broker not used anymore for message delivery. The replay is propagated along the old delivery path in the direction of the junction, from there it is sent along the new path to the new location of C where old notifications are delivered to the client eventually.

After termination the effect of the algorithm is that a relocating client effectively has bridged phases of disconnected operation, without losing notifications and with almost the same delivery guarentees as in the non-mobile case. We will explain how this is achieved throughout the next sections.

5.3.2 Prerequisites

The solution sketched in this chapter can be used in every environment that meets the following requirements. First, border brokers have to install and maintain a buffer for all notifications that are not yet delivered for a certain period of time in order to deal with disconnects. Second, the underlying routing infrastructure uses advertisements. Although not strictly necessary, the relocation effort is reduced substantially in that they guide the search for the old delivery path. Simple routing is assumed as routing strategy for now and extended later. Finally, border brokers or clients must have some means of detecting the new configuration that a client has entered the range of the broker. Some form of beacon or heartbeat is presupposed.

5.3.3 Algorithm Outline

We use a stepwise refinement of traditional subscription processing as described in Section 3.3 to devise the algorithm:

1. When reconnecting to a broker, subscriptions are automatically reissued so that clients do not need to re-subscribe manually.
2. The broker network configuration is updated to accommodate to client relocation rather than handling an (otherwise necessary) independent new (re-)subscription from a new location.
3. Notifications forwarded to the old location have to be replayed to the new one in order to bridge disconnectedness.
4. Delivery of new notifications has to be postponed until the replay is finished. In this way, moving does not influence the per-sender order of notifications, fulfilling the ordering requirement.

Basic case: single producer. Consider the scenario of Fig. 5.3(a). Client C is moving from broker B_6 to broker B_1 (step 1 in the figure). The local broker, which resides on the client, e.g., in form of libraries, is informed by the new border broker about its relocation, according to the prerequisites. It then reissues active subscriptions, which were previously forwarded through and recorded in the local broker anyway. By avoiding manual re-subscriptions of the client application, the first requirement of mobility transparency (cf. Req. 5.2.1) is achieved, i.e., the interface to the middleware is not changed.

In the second step, we enable the publish/subscribe middleware to relocate the client. The goal of the relocation process is to update the routing configuration and redirect the old delivery paths to

C to the new destination. During this process, the reissued subscription is propagated as usual in the direction of any received advertisement through B_2 and B_3 to broker B_4, setting up their routing tables. At B_4 the old and new path from producer P to client C meet (dotted and dashed line, respectively). Broker B_4 is aware of the junction because an entry of the old path of this subscription/client is already in its routing table.[2] When the routing table in the junction is updated, new published notifications will be delivered to the relocated client. Without assuming any knowledge about the old location of the moving client, the system is able to draw from localities in that only a portion of the delivery path is changed. Changes are limited to the smallest subgraph necessary for diverting routing paths, facilitating the timeliness/efficiency requirement which is only available with inherent middleware support.

The third step ensures completeness over phases of disconnectedness during movement. The junction broker B_4 sends a fetch request along the old path to B_6 following the routing table entries for the given subscription. All brokers along this path update their routing tables such that they are pointing into the direction the fetch originates from, i.e., B_4. Border broker B_6 as last recipient replays all buffered notifications. If delivered notifications are annotated with sequence numbers by the border broker, reissued subscriptions can in turn carry the last received number to qualify the replay. Note that replays are forwarded only in the direction of a specific subscription and do not mingle with other clients' data. After replaying the path from the old broker to the junction can be shut down by deleting the subscription's routing table entries as long as advertisement and routing entry point into the same direction; thereby excluding and stopping at the junction. In this way the notifications that passed the junction broker before its update are collected and sent towards the new location, ensuring the required completeness.

The last extension finally reorders the notifications so that the sender FIFO condition remains valid after relocation. The new border broker has to block and cache all incoming notifications that are to be delivered to the given client (not impeding communication of other clients) until the replay is finished. Of course, additional mechanisms like timeouts have to ensure that delivery is not delayed indefinitely. As with all buffering, consistency can always only be guaranteed for a predefined, finite amount of time or space.

Extension: multiple producers. We extend the previous example in order to cope with more than one producer. Let us assume a scenario like in Fig.5.3(b), figure on the right. The scenario is the same as before except client C was subscribed to more than one producer. The only change in behaviour of the process is that every broker B_i on the old path towards B_6 starting with B_4 has to check whether there is more than one advertisement for a producer matching filter F. If so B_i is a broker at a junction, like B_4, and hence the path towards B_4 must not be discarded. Only the part after passing the last junction can be deleted safely. In order to determine which is the last broker with a junction the algorithm uses the fetch request in one direction and the replay message in the other. On the way towards B_6 every broker with a junction towards a new producer replaces the broker ID in the fetch request with its own. In this example the sequence is B_4 (starting the process) and than B_5 when reaching the next junction at broker B_5. B_6 receives a messages where the broker ID is set to B_5 because no more junctions are on the path. The replay message on the other hand is used to delete the not needed appendix of the path towards B_1. As soon as B_5 receives a message from B_6 and the ID of B_5 equals the ID contained in the message, B_5 "knows" it is the last broker with an additional producer on the path towards B_6 and therefore cannot delete the routing table entry for (C, F) but has to modify it to point to B_4. Compared to the first example the difference is

[2] Subscriptions can be identified if simple routing is used. For covering and merging cf. Section 5.5.3.

obvious: While in the first scenario B_6 and B_5 could garbage collect (C, F), in the second scenario only B_6 can do so, while B_5 updates its routing table, yielding the correct and desired behaviour in both cases.

5.4 Algorithm Details

We now give the complete details of the algorithm for roaming clients which was presented by example in Section 5.3. The algorithm itself is given in two figures:

- Fig. 5.4 gives the algorithm for a border broker directly connected to a moving client
- Fig. 5.5 gives the algorithm for an inner broker made aware of moving clients and receciving messages from neighboring brokers.

5.4.1 Basic Case Analysis

By design, the mechanism we propose introduces a natural way of distributed caching. Hence, for every border broker connected to a client a caching data structure must be added.

We now analyse the different cases which the algorithm caters for and relate them to the standard behavior of the underlying REBECA system. The algorithm has to distinguish between four important situations:

1. A completely new client "wakes up" for the first time and submits some subscription to a broker B for the first time. This is slightly different from a new subscription issued after B has already received some other subscription from this client (i.e., knows already that the client is present).
2. The client was suspended (e.g., for saving energy) and has now resumed operation and wants to be set up properly by receiving events it has missed in the meantime but no location change occurred.
3. A client C connecting to a border broker B_{new} is a roaming client moving from one location to another (i.e., sending a subscription (C, F) with a sequence number of the last notification received for this subscription to broker B_{new}). Obviously, this has to be done whenever the client detects the change of context[3].
4. A client is suspended at one location and then moved to another location where it resumes operation. This situation is analog to a combination of the situations above and has not to be handled separately.

5.4.2 Algorithm Behavior

Given the four situations above, we now explain how the algorithm reacts.

[3] Please note that this is a rather complicated situation for an event middleware: as a mobile client usually cannot predict a change of broker before leaving its range (e.g., because it is just leaving a wireless network cell) it can only react to the new situation. For a "roaming-aware" client this means that it cannot unsubscribe to a producer at the old border broker before connecting to a new one. Hence, mobility in a publish/subscribe system is more likely a sequence of subscribe operations than a sequence of subscribe-unsubscribe-subscribe operations.

```
/** upon recv sub (C, F, num) via link L_C */
void receiveSub(C, F, num, L_C) {
  if (!RoutingTable.contains(C, F)) { // Client/Filter is unknown
    routeTable.add(F, C, L_C); // Allocate new RoutingEntry
    initializeCache(C, F); // Initialize cache for the client
    if (num > 0) {
      RoutingTable.setBlockingFlag(C, F, L_C, true); // wait until relocation
    }
    propagate(C, F, L_C); // propagate to all neighbors, except L_C
  } else { // client reconnects
    sendCache(C, F); // finish
  }
} // receiveSub()

/** upon recv fetch(C, F, num, B_prod) from B_j */
void receiveFetch(C, F, num, B_prod, B_j) {
  if (advertisement.numberOfMatches(F) > 1) {
    routeTable.update(F, C, L_Bj); // change RoutingEntry to B_j
  } else {
    routeTable.delete(F, C, RoutingTable.ALL); // remove all entries
  }
  send(new ReplayEvent(new FetchMessage(F, C, num, B_prod),
                                        [e_1, ..., e_n],
                                        B_j); // send replay back to B_j
} // receiveFetch()

/** upon recv replay(fetch(C, F, num, nil), [e_1, ..., e_n]) from B_j */
void receiveReplay(FetchMessage(F, C, num, B_prod),
                                [e_1, ..., e_n],
                                B_j) {
  prependToCache([e_1, ..., e_n], C, F);
  routeTable.setBlockingFlag(F, C, B_j, false); // unblock delivery
  // delivery resumes automatically
} // receiveReplay
```

Figure 5.4: Actions of a border broker receiving a message from a client C

```
/** upon recv of subscription (C, F, num) from B_j */
void receiveSub(C, F, num, B_j) {
  if (!RoutingTable.contains(C, F)) { // Client/Filter is unknown
    routeTable.add(F, C, B_j); // Allocate new RoutingEntry
    propagate(C, F, B_j); // propagate to all neighbors, except B_j
  } else { // the current broker is the junction
    brokerOldNext = routeTable.get(C, F); // remember old path
    routeTable.update(F, C, B_j); // update to new path
    send(new FetchMessage(C, F, num, B_current), brokerOldNext);
                                  // send fetch message along the old path
  }
} // receiveSub()

/** upon recv of fetch(C, F, num, B_prod) from B_j */
void receiveFetch(C, F, num, B_prod) {
  brokerOldNext = routeTable.get(C,F); // store old path information
  routeTable.update(F, C, B_j); // update to new path
  if (advertisements.numberOfMatches(F) > 1) {
      send(new FetchMessage(C,F,num,B_current, brokerOldNext);
                                          // replace last junction marker
  } else {
      send(new FetchMessage(C,F,num,B_prod, brokerOldNext);
                                          // keep last junction marker
  }
} // receiveFetch()

/** upon recv of replay(fetch(C, F, num, B_prod), [e_1, ..., e_n]) from B_j */
void receiveReplay(FetchMessage(C, F, num, B_prod),
                                  [e_1, ..., e_n],
                                  B_j) {
  brokerNext = routeTable.get(C, F);
  if (B_prod != null) {
      if (B_prod == B_current) {
        send(new ReplayEvent(new FetchMessage(C,F,num,null),
                                  [e_1, ..., e_n],
                                  brokerNext); // send fetch message
      } else {
        routeTable.delete(C, F, RoutingTable.ALL); // clean up
        send(new ReplayEvent(new FetchMessage(C, F, num, B_prod),
                                  [e_1, ..., e_n],
                                  brokerNext); // send fetch message
      }
  }
} // receiveReplay()
```

Figure 5.5: The algorithm for an inner-network broker receiving a message from broker B_j.

Wakeup of a new client. This case is already covered by the standard REBECA middleware: The subscription is added to the local routing table and the subscription (C, F) is forwarded in the direction of all known advertisements matching filter F. As a completely new subscription starts with the sequence number $num = 0$, the broker can separate this situation from situation 3. For details refer to the first block of statements of Fig. 5.4.

Resuming of a client at same border broker. This case is also covered by the standard event mechanisms provided by REBECA, as the broker knows the client and its subscriptions (the broker has some routing table entries for the client) and therefore event delivery simply could be resumed. Nevertheless, to adhere to the requirement of completeness, the broker has to instantiate a *cache* for each client and subscription (we propose a ring buffer data structure for enforcing the maximum number of cached notifications together with a Time-to-Live (TTL) mechanism for optimizing the utilization of buffer space). Whenever the client gets online again it is forced to re-issue its subscription together with the last sequence number it has received from a broker. The broker simply has to send all cached notifications to the client beginning with the next notification in the sequence (see first block of statements in Fig. 5.4).

Direct relocation. The main issues of the relocation process are: (i) tracking down the old location without explicit knowledge, as neither client nor new broker know the old broker's identifier, (ii) fetching any cached notification for the client from there, and (iii) the old broker has to receive an explicit sign-off for garbage collection. We have to distinguish between border and inner brokers for this case:

1. Border broker at new location:
   This is comparable to situation 2 above together with a new flag on a routing table entry (*inactive*) initiating buffering until further notice (see second and third block of statements in Fig. 5.4). Upon reception of a *replay* message, the cached notifications will be appended to the replayed ones and then delivered.

2. Any inner broker:
   For any given relocation process an inner broker can play one of three possible roles, either it is an ordinary broker on the new path from the producer to the consumer and has not encountered this particular subscription before, or it is the one broker "sitting" on the junction between the old and the new path, or one of the brokers on the path to the old location, somewhere between the broker at the junction and the border broker at the old location.

   - Ordinary broker:
     The same as every inner broker for situation 1 (see also first block of statements in Fig. 5.5).
   - Broker at a junction:
     This is a broker sitting on the junction between old and new path on the itinerary to a client[4]. The broker can determine this state for any incoming subscription by inspecting the routing table. Whenever a given subscription is already present, the broker is the first one on the path to the old location (with respect to the propagated subscription). We introduced a new inter-broker message type $fetch(C, F, num, B_i)$ (piggybacked to

[4] Note that such a broker always exists on the path to a particular producer due to the structure of the broker network as defined in section 3.3.3.

a normal subscription) to handle this situation. The message is sent along the old path for two reasons: (i) to initiate sending of cached notifications at the old location, and (ii) to determine the number of obsolete hops on the path which can be discarded after the client has left. Finally, the broker updates its routing table to point to the new location and resumes normal operation.

- Broker on the old path receiving a *fetch*:
  Whenever an inner broker receives a fetch request (see second block of statements in Fig. 5.5) obviously the broker is a hop on the path between the first junction and the old location of client C. It has to send the message further along the path after checking whether or not it has a path to another producer the client was subscribed to (multiple producer) and (i) if not, to simply update its routing table to point to the broker the message was received from or (ii) if so, to replace the broker identifier in the *fetch* request with its own, to indicate that the path to client C is not obsolete from this broker on and then to update the routing table like in (i).

- Broker on the old path receiving a *replay*:
  Whenever an inner broker receives a $replay$ message[5] it simply routes the message towards the new client location after checking whether or not it can discard the routing table entry for client C. To do so it checks the broker identifier B_{prod} enclosed in the message: (i) the identifier is not the same as the one of the current broker, i.e., the current broker is located some hops after the last junction, or (ii) the identifier is identical, i.e., this broker is located at the last junction. In case of (i) the routing entry can be discarded and in case of (ii) the broker identifier in the replay message is set to some well known value nil indicating that the process of freeing resources has terminated.

3. Border broker at old location receiving a *fetch* message:
   Whenever a border broker receives a $fetch$ message for some (C, F, num) from broker B_j it has to create a new replay message with all buffered notifications for (C, F) starting from $num + 1$ and send it in the direction of B_j. After this it can clean up und de-allocate all resources allocated to client C, i.e., has received an equivalent to an explicit “sign-off”.

Once the complete relocation process has terminated the algorithm asserts the requirements stated above, namely that (i) all notifications send to the old location (modulo messages discarded due to limited caches or expiration periods) will eventually reach broker B_{new}, (ii) broker B_{new} delivers all notifications to client C in correct order, i.e., respecting sender FIFO by sending redirected notifications first and then newly arrived ones, (iii) broker B_{old} at the old location will eventually detect the relocation of client C and can discard all subscriptions and caches for this client, i.e., has a simple and elegant criteria for garbage collection.

Client move after suspension. As the new broker cannot distinguish between a client newly powered on or one entering its range, this situation is already covered by situation 1, 2, and 3.

[5] This message type is part of the standard REBECA system and in this context indicates that the relocation process is finishing and on the “way back” to the new location of client C.

5.5 Discussion

In this section, we want to discuss certain aspects of the algorithm presented before. In the course of discussion, we show that the design choices made for the relocation of mobile clients are justified in order to meet the requirements stated in Section 5.2. Moreover, we argue that our approach is superior as, to the best of our knowledge, it is the only approach meeting all requirements identified. We show where and why apparantly comparable approaches fall short.

The requirements we focus on here are:

- completeness (cf. Requirement 5.2.2),
- ordering (cf. Requirement 5.2.3) and,
- responsiveness (cf. Requirement 5.2.4).

5.5.1 On Cache Management

The algorithm proposed above only uses mechanisms offered by the underlying publish/subscribe infrastructure, thus adhering completely to the publish/subscribe paradigm, which we consider desirable. However, every broker must maintain *per-subscription* caches, i.e., for every subscription issued by a client, a separate cache is initialized and maintained until the broker encounters one of two possible cases:

1. The cache is relocated to another location or
2. the cache expires due to a timeout.

Consequently, any border broker B_j, which is hosting n clients, has to maintain $caches_{B_j}$ caches, with

$$caches_{B_j} = \sum_{i=1}^{n} |sub_i|.$$

$|sub_i|$ is the number of subscriptions issued by client i at border broker B_j. Additionally, any local broker and border broker has to maintain a sequence number associated to each subscription. It should be clear that we face a trade-off between the resources available at a border broker and the requirement of completeness, as stated by Requirement 5.2.2. Although we assume border brokers explicitly *not* to be resource-limited, with a growing number of clients and subscriptions the available resources allocatable to a client are bounded. Thus, in a worst case scenario, completeness is not maintainable. A client suspended for too long must cope with the expiration of formerly allocated caches when the border broker assumed a client to be "dead" or the loss of some notification that were too old for delivery (expiration date in the notifications) or are overwritten in the cache when a circular buffer exceeds its capacity. Even if storage and resource constraints in the border brokers are not of concern, mobile clients may be disconnected for a long period of time in which more missed notifications are cached than the client can handle during replay. The possibly limited resources of mobile clients must be taken into account when designing cache sizes or limiting the replay by semantic filtering [HGM01].

However, clearly, our approach tries to avoid problematic scalability issues by using a fully distributed solution. Every border broker in the network with clients attached to it maintains local caches. Thereby, we try to avoid issues of a potential bottleneck as a central caching solution would most likely run into.

5.5.2 On Ordering and Completeness vs. Responsiveness

On connection of a client, a border broker has to determine whether a new client appeared and a relocation process is necessary, or a client reconnects after a suspend phase. If the client is unknown to the broker relocation is initialized for every subscription of client C separately. To do so, three fundamental phases of the algorithm are used: the propagation, the fetch, and the relocation phase. The first phase is basically identical to any common subscription processing. However, the second and third phases are unique to client relocation. In the second phase a *fetch* message is used to identify all paths from a *junction broker* to the old border broker. Moreover, an incoming *fetch* message also serves the purpose of an explicit move-out operation as it is assumed to be possible proactively in other approaches (cf. Section 3.5, e.g. JEDI), which, we argue, is unrealistic in mobile settings. During the relocation phase the old border broker replays notifications within the broker network from the old to the new location, using the freshly identified and marked path from old border broker to the new one. Along its way, the replay is used to facilitate garbage collection on those parts of the delivery path not needed any more for the new location of client C.

The summarized benefits of our solution are:

1. *No in-transit messages.* The *fetch* message sent along the path between junction and old border broker serves as a shutdown message or a "closing tag". After this message no other message related to client C is in transit to the old border broker. Hence, after receiving the *fetch* message for a subscription, the border broker instantaneously can start the relocation process.

2. *Sequence numbers.* As we maintain buffers in the old and the new border broker, together with a well-defined "switch-over" mechanism based on sequence numbers, no messages are lost.

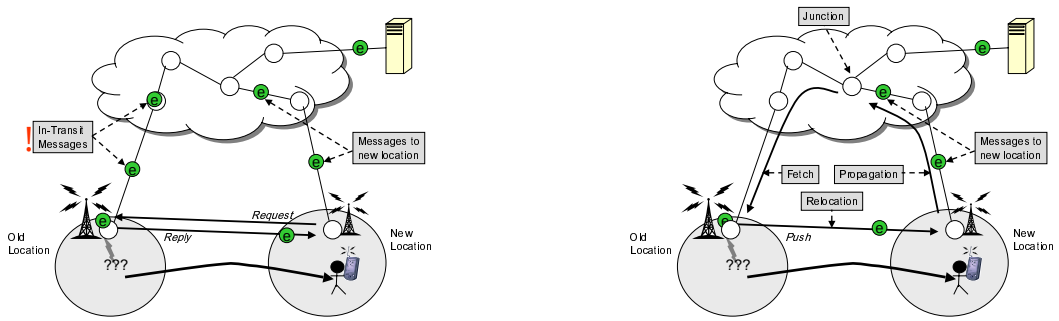
3. *Ordering.* As termination of the relocation process is guaranteed, the new border broker can withhold message delivery of new notifications until the relocation of old notifications is finished. By appending new notifications after older, relocated notifications, sender FIFO ordering is maintained.

4. *Full distribution.* The buffering scheme proposed in our algorithm is fully distributed. In general, this seems highly favorable in terms of responsiveness. Our algorithm assures that changes to the infrastructure are *minimal* and affect only the smallest possible subgraph of the routing network. Below we will discuss this issue in full detail.

5. *Simplicity.* The algorithm entirely adheres to the paradigm of publish/subscribe. Therefore, it can be based completely on mechanisms already implemented in REBECA or that can easily be added. This approach fosters implementability on top of an existing notification service.

While item 1. and 2. together guarantee completeness, item 3. states the matching of the ordering requirement. Naturally, this is only accurate taking the design of the underlying algorithm into account. The only requirement actually not fully tackled is the requirement of *responsiveness*. One might argue that by only using the publish/subscribe infrastructure for relocation, unnecessary delays in the relocation process are introduced. However, using *out-of-band communication* for speeding up the relocation process may introduce unwanted side-effects as we show below.

Shortcomings of alternative *out-of-band* solutions. The term *out-of-band communication* denotes any communication which is not done using the communication primitives and broker



(a) Pull (b) Push

Figure 5.6: Schematic behavior of out-of-band communication

network offered by the publish/subscribe infrastructure. The most important are direct request/reply and push-based protocols.

In a mobile scenario, two obvious solutions for *out-of-band communication* seem promising (cf. Fig. 5.6(a) and Fig. 5.6(b), respectively):

- *New location ⟶ Old location, request/reply.*
  Whenever a client connects to a new border broker B_i, it submits a unique and globally resolvable address of border broker B_j, responsible for the old location. The new border broker B_i subsequentially opens a direct channel to the border broker at C's old location and requests any notifications belonging to client C, using the client's ID and, per subscription, the last sequence number received. Broker B_j transmits the associated caches and unsubscribes from the information in question, if C was the only subscriber for this data. On the other hand, B_i, after having received the cache replay, subscribes to the same data at the new location.

  The prime advantage of the sketched solution is responsiveness, as, in general, a direct connection is supposed to have shorter latencies than a connection using the topology of the broker network. However, this solution violates Requirement 5.2.2 (completeness) and in some cases Requirement 5.2.3 (ordering). In Figure 5.6(a) the situation of *missing notifications* is depicted. A client connecting to a new border broker is requesting the cache for some client C. The border broker at the old location cannot make assumptions about messages *in transit*, i.e., which are in delivery somewhere in the broker network. Thus, to gain advantage over our solution in terms of relocation speed, the current *snapshot* of the cache must be transmitted to the requesting broker. Obviously, messages are lost whenever they are in transit in the part of the delivery path unique to the old location, i.e., after the *junction*.

  However, to avoid message loss due to in-transit messages another solution can be chosen. Border Broker B_j located at the client's old location might keep the connection to broker B_i open for forwarding of messages for C received after the relocation event. Unfortunately, avoiding the pitfall of in-transit messages this solution violates the ordering requirement. As the new broker subscribes to the same information as the old broker, due to the layout of the

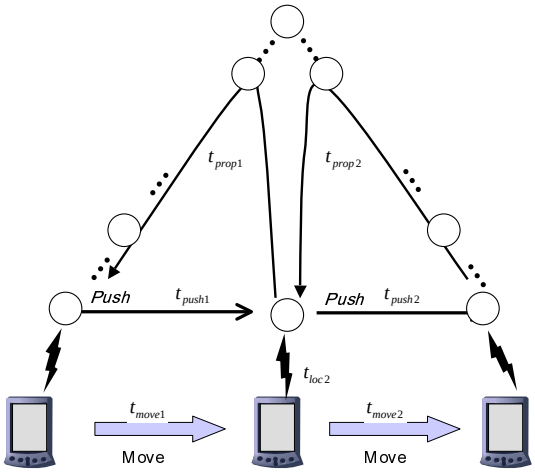


Figure 5.7: Erroneous push communication

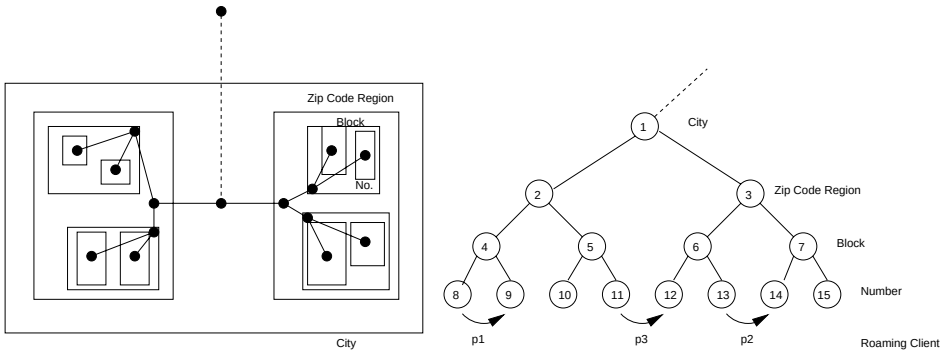
broker network and the unpredictability of latencies, the same information might be sent to both, the old and the new location. Without proper mechanisms for the detection of *duplicates*, this solution falls short, too.

- *Old location ⟶ New location, push-based.*
  A different approach, avoiding the pitfalls of the above approach, is to invert the communication relationship between old and new location and let the old border broker connect to the new location of client C. Here, the old broker is located using the first two phases of our algorithm as described in Section 5.3, i.e., the *propagation-* and *fetch phase*. Thereby ensuring that no in-transit messages are in the network and preparing the old delivery path for shutdown. The third phase (*relocation*) is done by establishing a direct communication channel between border brokers B_j and B_i. The caches for client C are then simply pushed to the new location. By prepending them to any notifications received and cached in the meantime, no ordering problems or message loss can occur. In the static case this approach assures the same guarantees as our algorithm presented above. But the sketched approach fails if location changes occur to often.

  As depicted in Figure 5.7, let a client C move from location L_1 to location L_2, taking time t_{move1}. At location L_2 it stays for time t_{loc2}, then moves on to location L_3, which takes the time t_{move2}. Relocation with push-based out-of-band communication fails if the following relation holds:

$$(t_{prop1} + t_{push1} + c_1) > (t_{loc2} + t_{move2} + t_{prop2} + t_{push2} + c_2), \text{ with } c_1, c_2 \in \mathbb{N}$$

  Where c_1 and c_2 are constants representing the accumulated processing time needed for relocation. The basic failure criteria is that the broker infrastructure needs some time for setting up the broker network before caches can be pushed. Should the relocation process from L_1 to L_2 take longer then the client stays at L_2 and relocation from L_2 to L_3 is fast, than L_2 might



(a) Hierarchical location model (b) Tree structure

Figure 5.8: Hierarchical location model of a city

already have reallocated resources for C when data from L_1 is pushed, resulting in message loss.

Neither approach, request/reply nor push-based, suffices to meet all requirements at the same time. Request/reply is the least favorable approach as message loss due to in-transit messages can be costly in case of incremental notifications, resulting in serious inconsistencies. We are aware of some notification services using this approach, but none using something similar to our proposed push-based approach. However, under certain constrains push-based relocation can be considered as a viable optimization of our approach, namely whenever movement speed can be assumed to be slow enough and garbage collection in inner brokers can be done externally.

On Message Complexity In this paragraph we will informally analyze and give a feeling for the message complexity of the proposed algorithm.

- *Best-case.* The most basic case is when a roaming client changes from one broker to a nearest neighbor in the broker network. In Figure 5.8(b) the change of a roaming client C from node 8 to node 9 can serve as an example. Then, the direct parent node in the broker tree acts as junction broker (node 4) and one of its direct sibling nodes is the old border broker of client C. Hence, exactly four messages are sent and the complexity class is $O(1)$.

- *Worst-case.* The largest possible distance between two brokers is constituting the worst-case scenario. In the network of brokers this is the two times the *height* of the spanning tree of brokers.

  DEFINITION 5.5.1 *Let $h(T)$ be the height of the spanning tree of brokers T, with $h(T)$ being the largest number of edges between the root node of tree T and a leaf node of T.*

  Obviously, then the worst case is $4h(T)$,

- $h(T)$ for the propagation phase,
- $h(T)$ for the fetch phase,
- $2h(T)$ for the relocation phase.

As we have depicted in Figure 5.8, hierarchical location models (Fig. 5.8(a)) can easily be mapped to a *well-balanced* tree structure (Fig. 5.8(b)), if a system is well-designed for the use with mobile clients and a notification service. The actual instantiation of a tree structure is dependent on the concrete real-world layout of a physical location. We use a binary tree as shown in Fig. 5.8(b), as a binary tree structure is the worst case assumption in such a scenario [6]. Then the worst-case analysis gives

$$O(n) = log_2(n),$$

with n being the number of brokers in the network. In general, given a well-balanced tree of order k, the message complexity converges against $O(n) = log_k(n)$.

- *Average-case.* The average case analysis is rather more complex. The key challenge is to model the impact of client mobility appropriately. Influencing factors for the expected number of messages to be sent in the average case are:
  - The pairwise probabilities of client movements from some border broker B_i to another border broker B_j.
  - The number of messages sent whenever a particular movement occurs.

Obviously, the instances of probabilities are dependent on the actual physical realities. A well-behaving user (usually) cannot walk through walls or "transport" to other locations instantaneously. Hence, to model the expected number of messages in the system we get:

$$E_{messages} = p_1 n_1 + p_2 n_2 + \ldots + p_m n_m, \tag{5.1}$$

where m is the total number of possible location changes, p_i is the probability that a location change i occurs and n_i the number of messages sent for this relocation event.

For the slightly simplified scenario of Fig. 5.8 this results in

$$\begin{aligned} E_{messages} \quad = \quad & 4\,p_1\,n_1 \\ & + 2\,p_2\,n_2 \\ & + p_3\,n_3 \end{aligned} \tag{5.2}$$

where p_1 is the probability of a change to the direct neighborhood (cf. Fig. 5.8(b)), p_2 a change affecting the next larger subtree of tree T, e.g., a change from one *Block* to another, and p_3 a larger change of location causing messages to be sent through the root node of T, inducing the largest message complexity. However, given an even distribution, we can evaluate equation 5.2 to:

$$E_{messages} \quad = \quad \frac{1}{2}n_1 + \frac{1}{4}n_2 + \frac{1}{8}n_3$$

[6] This is because the "fan-out" is restricted to two outgoing links only.

$$= \frac{1}{2}\underbrace{n_1}_{4\cdot 1} + (\frac{1}{2})^2\underbrace{n_2}_{4\cdot 2} + (\frac{1}{2})^3\underbrace{n_3}_{4\cdot 3}$$

$$= \ldots = 4\sum_{i=1}^{h(T)}(\frac{1}{2})^i i \tag{5.3}$$

$$= h(T)(1-(\frac{1}{2})^{h(T)}) - (h(T)-1) + (1-(\frac{1}{2})^{h(T)-1}) \tag{5.4}$$

As it can be seen in Figure 5.9(a), in the average the algorithm converges against a fixed number of messages to be sent for the relocation, hence in well-layouted systems the average message complexity can be in $O(1)$. However, when probabilities change, so changes the average message complexity. For demonstration purposes in Figure 5.9(b) we have included the average analysis if a system is designed especially inefficient. For a result as presented in Fig. 5.9(b) we had to assume that in half of all relocations the case with the largest possible number of messages sent occurs and the most "harmless" case only with a probability of $p_1 = \frac{1}{8}$ for our example of a tree of height $h(T) = 3$ (cf. "p_3" and "p_1"in Fig.5.8(b)). Then we get:

$$E_{messages} = 4\sum_{i=1}^{h(T)}(\frac{1}{2})^{h(T)+1-i} i \tag{5.5}$$

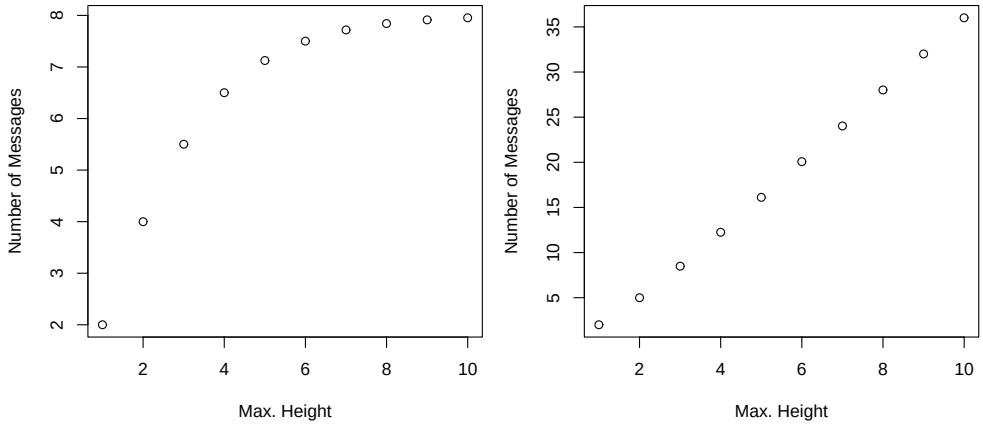
Obviously, the message complexity is $O(h(T))$, in this case, i.e., the number of messages sent is increasing linearly with the height of the tree. In case of a balanced tree, the tree increases in height with every doubling of brokers in the network[7]. Therefore, even in a non-optimal layouted system the algorithm remains well-behaving. But in most of the cases the result can be improved significantly simply by reordering the inner nodes of the broker network according to the probabilities of location changes observed. It must be noted that the inner nodes are not dependent on the actual physical layout of the "real" world, as the border brokers are. Adaptation of the broker network according to the relocation events as occurring in the physical world can yield a significant improvement in terms of message complexity.

5.5.3 On Possible Extensions

Covering. If covering instead of simple routing is used to establish the routing tables, the fetch phase of the algorithm has to be extended. Now, the junction is reached if an entry with a covering subscription $F' \supset F$ is already registered. At this point the delivery path to the new location is correctly built up, but we do not know whether the old location lies in the direction of F' or in the direction of the advertisements. The fetch phase is extended in that fetch requests are sent towards all advertisements and all covering subscriptions; it is a kind of flooding in the overlay network of matching producers and consumers of similar interests. Only one of the fetch requests will not get dropped and finally reach the old border broker. The replay has to be flooded in the same overlay network if no tunneling mechanisms, internal or external, are used.

Merging. The previously denoted extensions can also cope with a broker network that is based on merging. Only the number of potential covers increases, and hence the size of the flooded overlay network. Both covering and merging promise to increase routing efficiency, but on the other hand aggravate relocation management.

[7] For the assumed case of a binary tree.



(a) Well-layouted system

(b) Badly-layouted system

Figure 5.9: Average case analysis

Movement Speed. For simplicity reasons we assume that the client's movement speed is not too fast for the relocation process to terminate before the client moves again, i.e., the process always terminates at the correct broker. However, if re-subscriptions of the local broker are annotated with a relocation counter, which is reset after a successful replay, concurrent relocation processes can be identified and controlled in the middleware, avoiding the speed limit.

5.6 Summary

In this chapter we presented a solution to support mobility in existing publish/subscribe middleware.

A relocation algorithm is presented that facilitates physical mobility with location transparency, offering the possibility to transfer existing event-based applications to mobile scenarios as well as supporting mobility-aware applications that want to delegate the responsibility for relocation into the infrastructure. The algorithm seamlessly extends an existing content-based routing infrastructure, the REBECA notification service, to support non-interrupted, sender-FIFO ordered delivery of notifications to moving clients, which need not be aware of this extension. No central repository or control nor any communication outside of the publish/subscribe infrastructure is needed. On the other hand, applications can still benefit from the service's inherent benefits, like advanced routing algorithms. In the discussion of our solution we showed clearly that no other proposed or apparently comparable solution is supporting the publish/subscribe paradigm to the same extent as the solution we proposed. We showed that we can guarantee *transparency of mobility*, together with *sender FIFO ordering*, and *eventual completeness of notification delivery*. By analyzing best-, worst-, and average-case scenarios, we also showed that our solution is well-behaving in terms of message complexity. We also identified the key parameters for optimizing message complexity in any given environment, i.e., layout of the physical environment and logical layout of the broker network.

The presented solution for mobile clients in publish/subscribe systems transfers the characteristics of the publish/subscribe paradigm to mobile scenarios in an appropriate way. Loose coupling and drawing from notification delivery localities is explicitly supported. The underlying idea for extending REBECA was partly presented in [ZF03; FGKZ03].

6 Exploiting Uncertainty for Location-Dependent Notification Delivery

There is such a choice of difficulties,
that I own myself at a loss how to determine.

James Wolfe, 1727-1759, British General.

6.1 Introduction

In Chapter 4 we have laid the foundations to incorporate location as a first-class abstraction into the core of a publish/subscribe notification service. There, we introduced a reference location model, together with a precise definition of a location graph and its derivate, the movement graph. We made use of the location model not only to specify concrete location-dependent subscriptions and notifications, which is helpful for non-mobile but location-aware clients, we also specified the semantics of a special location marker `myLoc`.

We introduced `myLoc` as a means for *mobile* clients to express a standing request for location-dependent data while roaming around. A client that is interested in some location-specific information can express its interest by using the placeholder `myLoc` in the specification of the location-dependent part of a subscription. The infrastructure then is responsible for updating the placeholder `myLoc` according to any relevant position update of the client that was detected.

As we showed in Section 4.4.3, it is reasonable to place this functionality within the infrastructure to relieve a client from location-handling. The basic algorithm to facilitate location-dependent subscriptions in the notification service is shown in Figure 4.10 and Figure 4.11. But, due to the distributed nature of the underlying system certain problems arise when implementing the idea as discussed in Section 4.4.3 (cf. Fig. 4.7 on Page 70): either the solution introduces unwanted "black-out periods", where no notifications are delivered, or might lead to too much traffic for a client to process and possibly overloading a client. The former phenomenon occurs due to the implicit sequences of:

$$\texttt{subscribe}(\texttt{myLoc} = loc_1) \overset{movedTo(C,loc_2)}{\longrightarrow} \texttt{subscribe}(\texttt{myLoc} = loc_2), \texttt{unsubscribe}(loc_1)$$

The latter phenomenon occurs when *flooding* of location-dependent information is used, i.e., the location-dependency is mostly ignored and every information is available at all locations within a certain range. Then, `myLoc` basically is defaulted to `anyLoc`, as defined in Chapter 4 in Definition 4.3.2 on Page 61, and clients must filter out all unwanted information. However, as no change

of subscription is necessary, no blackout periods are experienced. Thus, the experienced responsiveness to location changes is significantly better than using sequences of "sub-unsub-sub".

Nonetheless, flooding is not only not efficient but also bears the potential to be "dangerous" for small devices (cf. Requirements 2.2.1 and 2.2.10) in terms of processing capacity and wireless network congestion. It is easily possible that a small device is rendered useless due to unwanted information overload caused by flooding.

We therefore want to explore the possibility to find a trade-off between the number of messages sent and the "smooth" hand-over from one location to another.

In this chapter we show how a *movement graph* can be exploited as a means to optimize the behavior of the algorithms introduced in Chapter 4. The results of this chapter were presented in [FGKZ03].

The goal of our approach is to achieve comparable responsiveness of the location hand-over "as if" flooding were used and, at the same time, avoid the drawbacks. By intelligently extending the basic mechanism of "sub-unsub-sub", we exploit context-information available about the *possible future locations* of a client to pre-subscribe to location-dependent data in a timely fashion. The effect is that the approach introduced in this chapter minimizes setup times in the infrastructure for a client and uses a form of "restricted flooding" for the optimization of responsiveness when a client moves around.

6.2 Basic Idea

We allow clients to specify subscriptions using the special marker `myLoc` in the same way as introduced in Chapter 4. However, as we have shown there, the "naïve" implementation using "sub-unsub-sub" sequences to update the routing tables for a particular client to the current location might lead to unwanted results. The least desirable effect for a client are blackout periods due to an unpredictable setup time. This does not only include the time it needs to actually update the routing tables to accommodate a new position of a client, but also the time it needs to propagate the new location sighting through the event broker network. The information must be propagated by the tracking system to a location service and then as `myLoc`-update notification through the network of brokers (cf. Figure 4.9 on Page 73 for details).

However, the overall goal is to significantly reduce the accumulated setup times such that a client experiences a frictionless change of location explicitly without a notable setup time after having changed from, e.g., the office to the conference room next door. The adaptation of the location-dependent subscription should take place instantaneously. This constitutes a principle of being subscribed to relevant location-dependent information "everywhere, at once", i.e., a semantics comparable to flooding.

The principal problem is that part of the setup time is determined by factors which are out-of-influence of a notification service. In Section 4.4.4.2 on Page 72, we explicitly separated the location tracking facilities from the event broker network. The notification service acts as client to location tracking as orthogonal concern. Moreover, the time and mode of location updates solely is subject to the (external) tracking system used and can take a significant amount of time. Thus, the total time it takes to propagate location changes through the broker network mainly is dependent on some external system.

Under this circumstances it is impossible to realize the needed quality of service for moving clients by adhering to a purely reactive model as proposed before. The basic idea now is to pre-subscribe to locations a client *possibly* will move to next. To do so, we rely on context information in the form

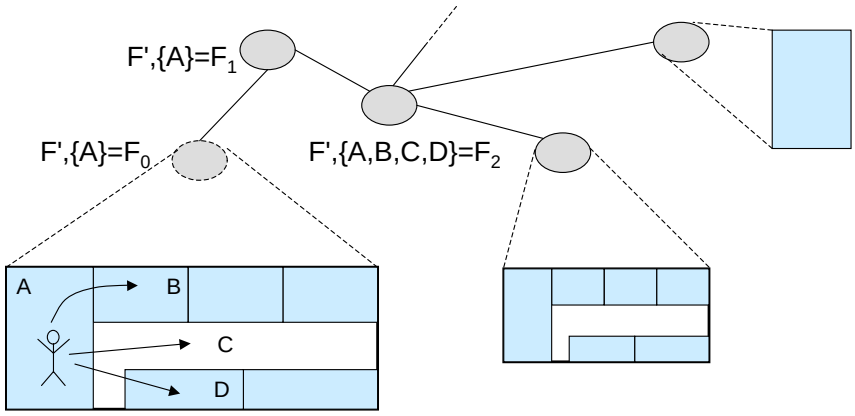


Figure 6.1: Uncertainty of myLoc in the broker network

of a *movement graph* of locations reachable from the client's current position. As we have shown in Section 4.4.4.2, a movement graph easily can be derived from available location information and the underlying location graph used.

We have illustrated the basic idea in Figure 6.1. A client at location A has issued a filter $F(C, \texttt{myLoc})$. Thereby, for the sake of the illustration, F' is the specification of application domain specific content and $\{l_1, \ldots, l_n\}$ denotes the set of locations myLoc evaluates to. The location-dependent filter issued then permeates the network and a routing path between a producer of information matching F and the client at location A is set up. This is exactly the semantics of myLoc in the first place. We now use knowledge about the reachable next locations for setting up alternative routing paths from producers of location-dependent data to the client. However, to prevent degeneration of this approach to simple flooding, we have to restrict the range of *possible* new locations to subscribe to *probable* new locations. The probability of a new location mainly depends on the (anticipated) movement speed of a client and the reachability of a location within a certain time-frame.

While the movement speed is dependent on the application domain and the anticipated clients, the latter can be taken from a pre-computed movement graph. Subscribing to probable new locations introduces *uncertainty* into the filter model of the publish/subscribe infrastructure, as routing paths for "virtual client" are established. Hence, information is delivered to locations the client eventually will not move to. One might characterize such behavior as a form of "restricted, adaptive flooding".

Effectively, by pre-subscribing to information at a probable new location, the following is realized:

- A client reaching a location, e.g., B in the example above, already receives location-dependent data for this new location, hence experiencing a frictionless change of location.
- The setup time for the new location is negligible.
- Once the location change is detected the *area of uncertainty* for a client can be updated and the notification service can unsubscribe to locations not reachable in due time and again pre-subscribe to probable new locations.

Throughout the following sections we will detail this basic idea and give an algorithm for the use in the broker infrastructure.

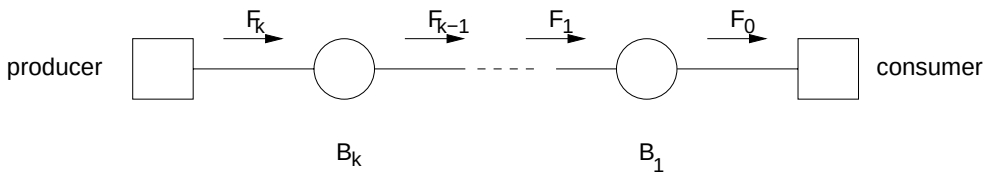


Figure 6.2: Filters along the path between producers and consumers.

6.3 Location-Dependent Filters for Logical Mobility

We now describe the algorithmic solution to the scenario where clients are locally mobile, i.e., they remain attached to a single border broker of the broker network. For location-dependent information delivery this is a typical situation, whenever location and application domain correlate. Typically, the application domain and thereby the location-dependent subscription change, when changing location at the "large scale". For example, the application domain is different when being in the office during daytime than the application domain used at home. Therefore, it is reasonable to assume that mobility is constrained to a single broker. However, we sketch a solution for location-dependent publish/subscribe together with roaming clients in the next Chapter 7.

6.3.1 Main Idea

Consider an arbitrary routing path between a producer (publisher) and a consumer (subscriber). This path consists of a sequence of brokers $B_1, B_2, \ldots, B_{k-1}, B_k$ where B_1 is the local broker of the consumer and B_k is the local broker of the producer (see Figure 6.2). Assume the consumer has issued a location-dependent subscription F. Using the "usual" content-based routing algorithms, the current value $\tilde{F}$ of F would permeate the network in such a way that the filters along the routing path allow a matching subscription which is published by the producer to reach the consumer. Formally, the filters $F_1, F_2, \ldots, F_k$ along the links between the brokers should maintain a set-inclusion property

$$F_k \supseteq F_{k-1} \supseteq \ldots \supseteq F_2 \supseteq F_1 \supseteq F_0 = \tilde{F}$$

at all times.

If F is the only active subscription in the network or simple routing is used and if the subscription has permeated the network, the above formula can be simplified to

$$F_k = F_{k-1} = \ldots = F_2 = F_1 = F_0 = \tilde{F}.$$

As being remarked above, for every change of location, i.e., for any new value $\tilde{F}$ of F, a new subscription must flow through the network towards the producers. Notifications which are published in the meantime for the new location of a client might go unnoticed due to the latency of location detection and filter setup.

In a nutshell, the idea of the proposed scheme is to always have the local broker of the consumer do perfect client-side filtering (i.e., set $F_0 = \tilde{F}$), but to let possible future notifications reach brokers which are nearer to the consumer so that their delay to reach the consumer is lower once the consumer switches to a new location. Effectively, we want to maintain a certain degree of uncertainty of a

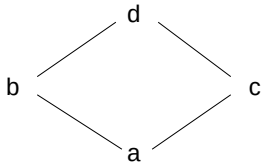


Figure 6.3: Movement graph defining movement restrictions of a consumer.

client's location in such a way that, once a client actually changes location, the filters for the new location are already set up.

More precisely, filter F_1 would be devised in such a way that it contains F_0 but also all notifications which should reach the consumer for any new consumer location which could be reached within "one step". Similarly, F_2 contains all notifications of F_1 but also those notifications which are possibly interesting if the consumer takes two steps. In a sense, the generality of the filters increases with the uncertainty of the future location: for a broker, there is more certainty about what will be interesting to the consumer in the future the closer the broker is to the consumer.

Let T denote the set of time values, which for simplicity we will assume to be the set of natural numbers $\mathbb{N}$. Let L denote the set of all consumer locations. Then we define a function *loc* : $T \rightarrow L$ which describes the movement of the consumer over time. For example, for a very simple location set $L = \{a, b, c, d\}$ a possible value of *loc* is $\{(1, a), (2, b), (3, d), \ldots\}$ meaning that at time 1, the consumer's location is a, at time 2 it is b and so on. Thereby, we have a means to describe a particular trajectory of a consumer.

We assume that *loc* is subject to movement restrictions as reflected by a movement graph (cf. Section 4.4.4) . In effect this defines a maximum speed of movement for the consumer. For example, such a restriction could result in a movement graph such as the one depicted in Figure 6.3. The graph formalizes which locations can be reached from which locations in one movement step of the consumer. Here, one movement step corresponds to one time step.

Given the function *loc* and a movement graph, we define a function *ploc* : $L \times \mathbb{N} \rightarrow 2^L$ of possible (future) locations (the notation 2^L denotes the powerset of L, i.e., the set of all subsets of L). The function takes a current location x and a number of consumer steps $q \geq 0$ and returns the set of possible locations which the consumer could be in starting from x after q steps in the movement graph.

Since a possible move of the consumer always is to remain at the same location, for all locations $x \in L$ and all $q \in \mathbb{N}$ we should require that

$$\textit{ploc}(x, q) \subseteq \textit{ploc}(x, q + 1). \tag{6.1}$$

Taking the example values from above, possible values for *ploc* are as follows:

$$\begin{aligned} \textit{ploc}(a, 0) &= \{a\} \\ \textit{ploc}(a, 1) &= \{a, b, c\} \\ \textit{ploc}(a, 2) &= \{a, b, c, d\} \end{aligned}$$

Now, if the consumer is at location a, for example, every broker B_i along the path towards a producer should subscribe for $\textit{ploc}(a, q)$ for some q, which is an increasing sequence of natural

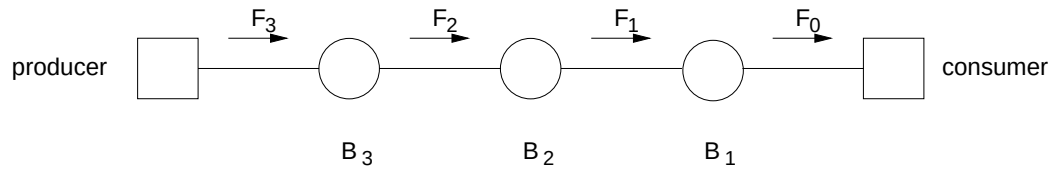


Figure 6.4: Network setting for the example.

t	$x = a$	$x = b$	$x = c$	$x = d$
0	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$
1	$\{a, b, c\}$	$\{a, b, d\}$	$\{a, c, d\}$	$\{b, c, d\}$
2	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$
3	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$

Table 6.1: Values of *ploc*(x, t) for the example setting.

numbers depending on i and the network characteristics. If the time it takes for a broker to process a new subscription is in the order of the time a client remains at one particular location, then the individual filters F_i along the sample network setting in Figure 6.4 should be set as $F_i =$ *ploc*(a, i), e.g., $F_0 =$ *ploc*$(a, 0) = \{a\}$, $F_1 =$ *ploc*$(a, 1) = \{a, b, c\}$ and so on. This requirement should be maintained throughout location changes by the consumer. For example, whenever a consumer moves from an old location loc_1 to a new location loc_2, the corresponding border broker eventually declares the new location by changing the location-dependent part of filter F_0 for client-side filtering from the old to the new location. All brokers concerned update their routing table appropriately.

In general, broker B_i sends a message with the new location to B_{i+1} instructing it to change F_i from *ploc*(loc_1, i) to *ploc*(loc_2, i) and consequently to update the routing table by removing certain locations and adding new locations. Removing and adding new locations corresponds to unsubscribing and subscribing to the corresponding filters. For our implementation using the REBECA notification service, common administration messages can be used to do this. Note that Equation 6.1 guarantees the subset relationship, which should always hold on every path between producer and consumer.

6.3.2 Example

As an example, consider the value of *loc* where at time 1 the client is in location a, at time 2 at b and at time 3 at d in the movement graph depicted in Figure 6.3. Table 6.1 gives the values of *ploc* for all locations and the first four time instances. For $t = 0$ the value of *ploc* is equal to the current location. For $t = 1$ it returns all locations reachable in one time step in the movement graph, etc.

Now assume again the setting depicted in Figure 6.4. The values of Table 6.1 directly determine the filter settings for $F_0, \ldots, F_3$ as shown in Table 6.2. At time $t = 1$ the client moves to location b. This means that F_0 changes from $\{a\}$ to $\{b\}$ and that F_1 must unsubscribe to c and subscribe to d, yielding $F_1 = \{a, b, d\}$. At time $t = 2$ the client moves to d, causing F_0 to change to $\{d\}$ and F_1 to

time t	F_3	F_2	F_1	F_0
0	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{a,b,c\}$	$\{a\}$
1	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{a,b,d\}$	$\{b\}$
2	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{b,c,d\}$	$\{d\}$

Table 6.2: Values of filters in example setting.

ploc(x,t) for global *sub*/*unsub*				
t	$x=a$	$x=b$	$x=c$	$x=d$
0	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$
1	$\{a,b,c\}$	$\{a,b,d\}$	$\{a,c,d\}$	$\{b,c,d\}$
2	$\{a,b,c\}$	$\{a,b,d\}$	$\{a,c,d\}$	$\{b,c,d\}$
3	$\{a,b,c\}$	$\{a,b,d\}$	$\{a,c,d\}$	$\{b,c,d\}$

ploc(x,t) for flooding				
t	$x=a$	$x=b$	$x=c$	$x=d$
0	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$
1	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{a,b,c,d\}$
2	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{a,b,c,d\}$
3	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{a,b,c,d\}$

Table 6.3: Values of *ploc*(x,t) for trivial *sub*/*unsub* implementation (top) and flooding with client-side filtering (bottom).

unsubscribe to a and subscribe to c. All other filters remain unchanged.

The example nicely shows that the method does some sort of "restricted flooding", i.e, all notifications reach broker B_2 but from there the uncertainty is restricted and so is the flow of notifications forwarded by B_2. In fact, the method described above using the *ploc* function can be regarded as an abstraction of both "trivial" implementations discussed in Section 3.4 (i.e., both implementations are instantiations of our scheme), as we explain in the following section.

6.3.3 Adaptivity

The example setting above assumes that processing a new subscription by a broker takes about as long as a consumer stays at one particular location. Obviously, it will usually take much less time to process a subscription even if slow or wireless network connections are used (user movement will be in the order of seconds while network delay will be in the order of milliseconds). We now present a scheme that adapts the level of "buffering" in the network to the average movement time of the client. Our algorithm satisfies this form of adaptivity. The details of our algorithmic solution can be found in Section 6.3.4 of this thesis and in [FGKZ02], accordingly.

In the following, we denote the average time a client remains at one location by Δ and the time it

t	$x = a$	$x = b$	$x = c$	$x = d$
0	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$
1	$\{a, b, c\}$	$\{a, b, d\}$	$\{a, c, d\}$	$\{b, c, d\}$
2	$\{a, b, c\}$	$\{a, b, d\}$	$\{a, c, d\}$	$\{b, c, d\}$
3	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$

Table 6.4: Values of *ploc*(x, t) for the example setting with concrete timing values.

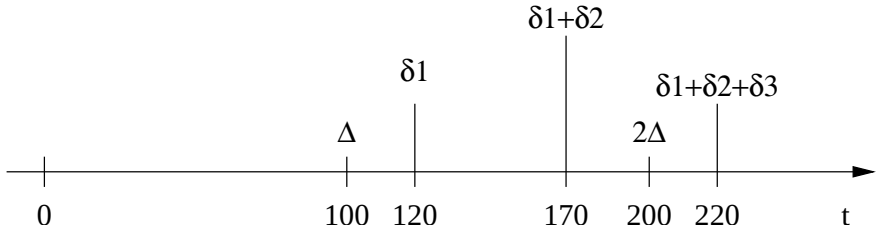


Figure 6.5: Estimating *ploc* steps with respect to concrete timing bounds.

takes to process a sufficiently large batch of *sub*/*unsub* messages between brokers B_i and B_{i+1} by δ_i. If the client moves very slowly, meaning that the sum of all δ_i is still less than Δ, we would like the scheme to behave like the trivial *sub*/*unsub* solution. For the example setting from the previous section this would mean that *ploc* has values like in the top part of Table 6.3 (note that the algorithm always has to provide information for "the next" user location to maintain the semantics of flooding). On the other hand, if the client moves very fast and Δ is much smaller than δ_1, the method should revert to flooding (i.e., *ploc* values like in the bottom part of Table 6.3).

If Δ is neither very large nor very small, what values should *ploc* acquire? The idea is to relate multiples of Δ to the increasing sum of the δ_i as follows: Whenever the sum of δ_i results in a value larger than the next multiple of Δ then the value of *ploc* must "take a step". As an example, assume the following values (all in milliseconds): $\Delta = 100$, $\delta_1 = 120$, $\delta_2 = 50$, $\delta_3 = 50$, $\delta_4 = 20$. Now consider Figure 6.5 where the sums of these values have been put on a single scale. The *ploc* value for client-side filtering (F_0) is fixed to the current location of the client. Since it takes longer for the brokers B_1 and B_2 to process a location change than the client moves, the system must insert a level of buffering at this point, i.e., *ploc* must cater for one additional step of uncertainty at this stage.

Considering that $\delta_1 + \delta_2 < 2 \cdot \Delta$, a location change can be processed fast enough between B_2 and B_3 so that no additional buffering is necessary at this point. However, the sum $\delta_1 + \delta_2 + \delta_3 > 2 \cdot \Delta$, and so *ploc* must have one additional step between B_3 and B_4. The resulting values in the example setting for *ploc* are shown in Table 6.4.

6.3.4 The Algorithm Proper

For a given location graph $Loc = (L, C)$, with L being the set of possible locations of some client X and C the set of edges denoting the relationships between locations, a movement graph $M = (L, E)$

```
/** upon receiving new location−dependent subscription F from X
 * with current location loc and given Δ via link L_n do */
void receiveSub(X, F, loc, Δ, L_n) {
  Entry e = ldfTable.createNewEntry(); // allocate new entry e in ldfTable
  ldfTable.add(e.setFilter(F,X,0)); // initialize e with F, X, and 0
  routeTable.add(X,F[L/{loc}]); // subscribe to F[L/{loc}]
  propagate(X, F[L/{loc}], Δ, 0, L_n); // and send (X, loc, Δ, 0) to
                                       // all neighboring brokers except L_n
}
/** upon location update for Client X:
 * moved from loc_old to loc_new given Δ do */
void receiveLocUpdate(X, loc_new, loc_old, Δ) {
  for (all F ∈ ldfTable referring to X) {
      routeTable.remove(F[loc/ploc_X(loc_old, 0)]); // unsub to old location
      routeTable.add(F[loc/ploc_X(loc_new, 0)]); // sub to new location
      propagate(X, loc_old, loc_new, Δ, 0) // propagate to all
                                           //neighboring brokers.
  }
}
/** upon receiving an unsubscription for
 * location−dependent filter F from X at location loc do */
void receiveUnsub(X, F, loc) {
  routeTable.remove(F[L/{loc}], X) //unsubscribe to F[L/{loc}]
  ldfTable.remove(F, X); //de-allocate entry containing
                         // F in ldfTable
  propagate(X, loc, F); // send (X, loc, F) to all neighboring brokers
}
```

Figure 6.6: Algorithm for the border broker B_X of client X.

can be generated. L holds the same locations as in Loc and E is the set of "movement edges" between the locations. This results in a graph as shown in Figure 6.3. As we mentioned before such a graph can be generated and deployed at setup time of the publish/subscribe system. We assume that all brokers know *ploc*.

Data structures. Every broker has an additional routing table data structure `ldfTable` where location dependent filters are stored in their original form, i.e., with the marker `myLoc` uninterpreted. In general, we assume that $F_X \equiv (V, (loc \in L))$, where V is an arbitrary sequence of name/value pairs without references to `myLoc`. We instantiate F into $\tilde{F}$ by simply replacing loc with some set of locations L'. We denote this instance as $F[L/L']$.

For every entry in `ldfTable` the originating client is stored (i.e., X) and a natural number *oldstep*.

Every broker B_i maintains a network statistic about the average network delay $\delta_{B_i \to B_j}$ for sending messages from B_i to B_j. Every client X maintains a statistic of the average time Δ he remains at a specific location.

```
/** upon receipt of (X, F, loc, Δ, dist) from B_j do */
void receiveSub(X, F, loc, Δ, L_n, dist) {
  dist := dist + δ_{B_i→B_j}; // calculate dist
  step := ⌈dist/Δ⌉; // calculate "uncertainty"
  Entry e = ldfTable.createNewEntry(); // allocate new entry e in ldfTable
  ldfTable.add(e.setFilter(F,X,0)); // initialize e with F, X, and step
  routeTable.add(X,F[L/ploc_X(x, step)]); // subscribe to
                                  // uncertainty area F[L/ploc_X(x, step)]
  propagate(X, F, loc, Δ, dist, B_j); // and send (X, F, loc, Δ, dist) to
                                  // all neighboring brokers except B_j
}

/** upon receipt of (X, x, y, Δ, dist) from B_j do */
void receiveLocUpdate(X, x, y, Δ, dist, B_j) {
  dist := dist + δ_{B_i→B_j}; // calc. distance
  for (all filters F ∈ ldfTable referring to X) {
    newstep := ⌈dist/Δ⌉; // calc. uncertainty
    routeTable.remove(F[L/ploc_X(loc_old, oldstep)], X); // unsub
                                                  // old locations
    routeTable.add(F[L/ploc_X(loc_new, newstep)], X; // sub new locations
    oldstep := newstep;
    ldfTable.update(X,F,oldstep); // store oldstep in ldfTable
    propagate(X, F, loc_old, loc_new, Δ, dist, B_j); // send
                  // (X, loc_old, loc_new, Δ, dist)
                  // to all neighboring brokers except B_j
  } // end for
}

/** upon receiving unsub (X, loc, F) from B_j do */
void receiveUnsub(X, loc, F) {
  routeTable.remove(F[L/ploc(loc, oldstep]); // unsubscribe to
                                          // F[L/ploc(x, oldstep]
  ldfTable.remove(F, X); // de-allocate entry for (F,X) in ldfTable
  propagate(X, loc, F, B_j); // send (X, x, F) to all neighboring
                          //brokers except B_j
}
```

Figure 6.7: Algorithm for the broker B_i receiving a message from broker B_j.

Algorithm. The algorithm for a local broker B_X of some client X is depicted in Figure 6.6. If a client issues a new location-dependent subscription F it is entered into the table *ldfTable* and the local routing table is updated by subscribing to the proper instance $\tilde{F}$. Information about this new filter is forwarded through the network within the algorithm for the other brokers (see Figure 6.7). During this process, all other brokers allocate an appropriate entry in their *ldfTable* and subscribe to the "right" instance of F given the current distance from X. Note that during the propagation of the new filter through the network the value of *dist* continuously sums up the network delay along the path. This value determines the "step" of the *ploc* function which is used to instantiate F correctly.

When a client changes location from loc_{old} to loc_{new}, all the brokers similarly update their routing tables by taking the information about the changed location, unsubscribing to the old filter and subscribing to the new correct instance of F. In doing so, the distance *dist* is recalculated and may also lead to changes in how F is instantiated.

From the algorithm it is obvious that the information about a new location-dependent subscription (and about every location change) necessarily permeates the *entire* network of brokers. But in the case of a dynamic network environment it is not possible to avoid this behavior. To see this, consider a client moving from location loc_{old} to loc_{new} and assume that the network behavior is the same in the entire network except that a network link between two very far away brokers B_j and B_i has suddenly become very slow so that it is necessary for B_i to increase the step in the *ploc* function and subscribe to "more" locations than before. But to do this, information about y is needed at B_i.

6.3.5 Informal Analysis

We now analyze our algorithm quantitatively. The main question we pose is how much network traffic our algorithm can save compared to flooding. The answer to this question depends on many different parameters. For our informal analysis, we calculate the total number of messages processed for a set of common network scenarios and derive some conclusions from these numbers.

The base scenario we consider consists of a publish/subscribe system which is built around a backbone of event brokers. The brokers within the backbone are connected with high speed communication links on which the network delay δ_f is low. However, clients are attached to the backbone by very slow communication links that have a high network delay δ_s. In our case we chose $\delta_f = 10ms$ and $\delta_s = 600ms$ and assume that they do not change.

To ease calculations, the broker backbone is assumed to have the structure of a tree with degree b and h levels (see Figure 6.8). This simplification is justified based on our discussion about location graphs and their impact on the layout of the broker network in Section 4.4.

However, for comparing the algorithm with flooding, scaling the producers is more important than the consumers. In fact, for flooding, each message produced crosses every network link in the broker network. How many consumers actually "listen" to those messages is unimportant. Hence, we assume that each border broker serves p clients which will play the role of notification producers in this scenario. In our case we set $b = 3$, $h = 4$ and $p = 10$. Because of the tree structure, we can calculate the total number of brokers n_b as $1 + \sum_{i=0}^{h} b^i$.

For the sake of the calculation, there is exactly one consumer in the system which is attached to a border broker at the root of the broker tree. This consumer issued a location-dependent subscription. We assume that the size of the set L of possible locations is 100 and that, with every possible step of the client, the number of possible locations is multiplied by some factor s. In our case, we assume $s = 4$. We assume that the user changes location every Δ seconds.

Producers generate r notifications per second which are uniformly distributed over the set L of possible locations.

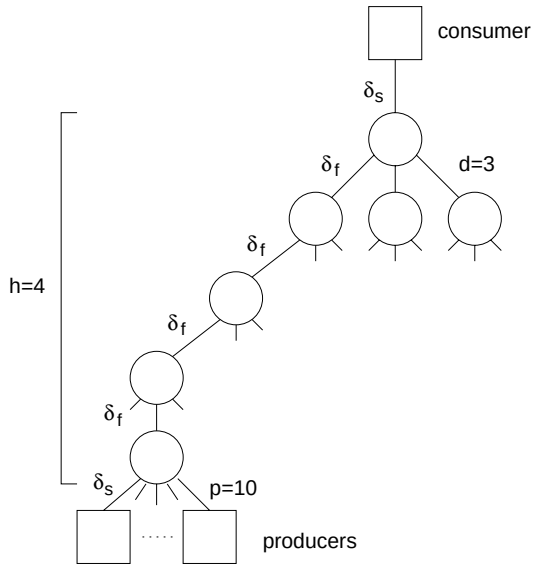


Figure 6.8: Sample scenario analyzed.

For every particular choice of values for these parameters, we calculate two values:

- $N_f(t)$: the total number of messages generated in the network if flooding is used as basic routing technique.
- $N_n(t)$: the total number of messages (notifications and control messages) generated in the network if our new algorithm for location-dependent filters is used.

We calculate these values for two different types of user movement: slow movement ($\Delta = 10s$) and fast movement ($\Delta = 1s$).

As flooding basically defaults `myLoc` to `anyLoc`, in this case the user movement is unimportant in the calculation of the total number of messages processed over time. Let n_{bb} denote the number of border brokers in the system serving producers. Due to the tree structure of the system, $n_{bb} = b^h$. Since every local broker has p clients (producers) attached to it, the total number of producers n_c in the system calculates to $n_c = n_{bb} \cdot p$. Every producer emits r notifications per second. Hence, the number of notifications produced per second $n_{ps} = n_c \cdot r$. Since every produced message has to cross every link, we can calculate

$$N_f(t) = n_{ps} \cdot n_{links} \cdot t$$

where, due to the tree structure, the number of links between brokers is $n_{links} = n_b - 1$.

The calculation of a similar formula for our algorithm depends on Δ. With $\Delta = 1s$ and due to the values for δ_s and δ_f, our algorithm must "buffer" one step of the movement graph per slow link, i.e., the border broker of the consumer receives all notifications which are one step away from the current location (i.e., a fraction of $s/|L|$ of all produced notifications). Also, the broker following the

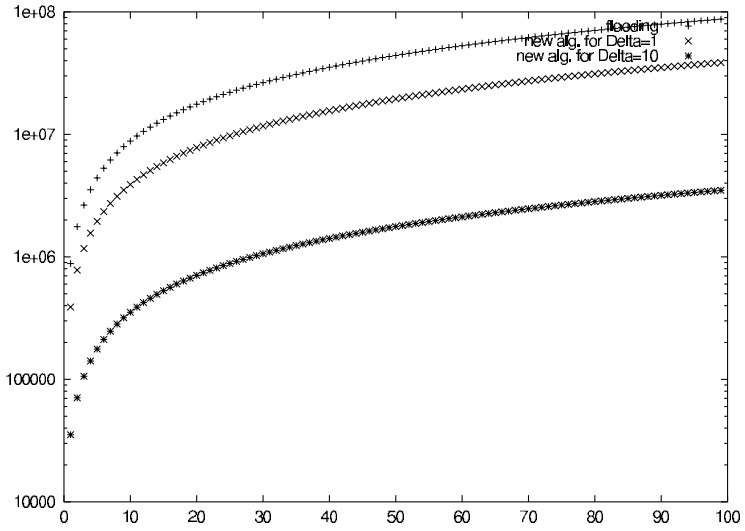


Figure 6.9: Total number of messages generated for flooding and two scenarios of the new algorithm. The values taken are $\delta_s = 600ms$, $\delta_f = 10ms$, $b = 3$, $h = 4$, $p = 10$, $|L| = 100$, $r = 1$, $s = 4$, $n_b = 364$, $n_{bb} = 243$, $n_c = 2430$, $n_{ps} = 2430$, $n_{links} = 363$, $n_{rest} = 120$.

border broker of the consumer must receive all notifications two steps away of the current consumer location (i.e., a fraction of $s^2/|L|$ of all produced notifications). In the case $\Delta = 10s$ the latter buffering is not necessary. This is where our algorithm plays to its strength.

For $\Delta = 10s$ we end up with

$$N_n(t) = n_{ps} \cdot n_{links} \cdot \frac{s}{|L|} \cdot t + n_{links} \cdot \lfloor \frac{t}{10} \rfloor$$

where the first part of the sum corresponds to the restricted flooding (the fraction of $s/|L|$ messages crosses every link) and the second part corresponds to the control messages introduced by our algorithm. Since control messages about location changes flood the network, their number must be treated like in flooding. However, a new control message is generated only once in $\Delta = 10s$.

For $\Delta = 1s$, we must take into account that a fraction of $s^2/|L|$ notifications crosses the first link from the border broker to the next broker in the network. Hence, we multiply every produced message with this fraction together with the number of links starting from border brokers (n_{bb}). To this we add the number of remaining links $n_{rest} = n_{links} - n_{bb}$ times the original fraction of $s/|L|$ of produced messages. The second part of the sum is again to account for the control messages, which flood the network once per second.

$$N_n(t) = \left[n_{ps} \cdot n_{bb} \cdot \frac{s^2}{|L|} + n_{rest} \cdot n_{ps} \cdot \frac{s}{|L|}\right] \cdot t + t \cdot n_b$$

The results of our calculations are depicted in Figure 6.9 (note that the y axis has a logarithmic scale). It shows that for the given settings and the two different scenarios the new algorithm imposes

a smaller number of total messages than flooding. The graph confirms that a slower user movement can save more messages.

6.4 Summary

In this chapter we presented an extended algorithm for location-dependent notification delivery that optimizes the "basic" version as presented in Chapter 4. As we have shown there, some problems of responsiveness versus completeness, i.e., the experienced blackout-periods when roaming, arise. However, the algorithm presented in this chapter has the same basic characteristics as the algorithm proposed in Chapter 4, but exploits information about the location and the possible movement of a client in the future to implement a semantics of "restricted flooding." Even more, the mechanism tries to achieve the experienced responsiveness of "real" network flooding of producer data to clients. On the other hand, it also tries to avoid the drawbacks of flooding, i.e., too much unwanted traffic "floating" through the network. This is especially important for resource-limited devices that would suffer significantly from too much data to filter and process locally.

To achieve this goals, we showed how an area of uncertainty of a client's location and movement can be exploited. By taking into account the whereabout of a client we can provide a setup of the broker network such that the necessary time to adapt to an actual change of location is minimized as if flooding was used. At the same time a movement graph is used to restrict this semantics of flooding to those locations that are probable for a client to reach in due time. Hence, flooding is limited to a certain area of the physical space.

Finally, we analyzed the proposed mechanism quantitatively in network settings, which are both, conservative enough for a concrete scenario and diverse enough to present a sound approach to identifying the factors that determine the potential of the algorithm. This way we have shown that a significant amount of traffic can be saved when compared to the "naïve" solution of network flooding.

7 Decoupling in Space and Time

Histories make men wise; poets, witty; the mathematics, subtile;
natural philosophy, deep; moral, grave; logic and rhetoric, able to contend.

Francis Bacon, English philosopher, essayist, and statesman (1561-1626)

7.1 Introduction

In this Chapter we address what in Section 2.2.4 was identified as being an important requirement for mobile environments: *decoupling in space and time*.

In Chapter 5 we laid the foundations for decoupled operation in the context of mobility support within the publish/subscribe infrastructure: buffering of notifications in the broker network to cater for disconnection and – more importantly – reconnection of mobile clients. Thereby, two goals were achieved: (a) the relocation functionality arbitrates the actual location of a client with respect to the broker network and (b) the buffering scheme we introduced facilitates decoupling over periods of disconnectedness and hence, time. The client does not have to be connected when a notification is reaching the last hop in the infrastructure and should be delivered to the device.

However, in this chapter, we broaden the notion of decoupling. In the remainder of this chapter we sketch how a mobile, roaming client can have access to location-dependent notifications that were already delivered in the past when the client was not connected to the current environment. Thereby we introduce an effective means to subscribe *into* the "past". The underlying idea was introduced in [CFH+03] and implementation details can be found in in [Gue04]. By making it possible to access information that has already been delivered in the past, we also decouple consumers and producers in space. The client does not have to be physically attached to a certain broker in order to "receive" certain location-dependent information at the time the information is propagated through the network. This introduces a certain degree of persistence of information, on the one hand, and extends the notion of location-dependent information delivery beyond the boundaries of a single broker as introduced in Chapter 6, on the other hand.

Problem statement. The observation we make is that a client needs a certain number of notifications within the flow of data in order to bootstrap properly. This is common design practice in static systems. But, an inherent assumption made in the design of static systems is that the participants of such a system, i.e., consumers and producers, are "online" constantly. Given this, a system usually simply needs a certain *settling time* of the clients to reach a stable state of operation.

Contrary to the classical model of distributed systems, nomadic and mobile computing systems introduce a significant level of client dynamics. Mobile clients join and leave an environment constantly and rather unpredictably. Any new client then needs some amount of information to reinitialize or adapt to the new setting. Moreover, as the client usually is "roaming", it stays only for a

certain amount of time within the current environment. This poses the need to explore possibilities for keeping the settling time for mobile clients as short as possible in order to allow for proper operation at the current location. Even worse, it might be impossible for a client to simply "stop" and wait for a while until the needed information is published as part of the information flow. A mobile client might have to make decisions based on location-dependent data which has to be available as soon as the client reaches a certain environment.

Example. Consider for instance the following scenario: a driver assistance application wants to warn a driver when the crusing car approaches a red traffic light. Therefore, the application has to access location-dependent data about the *current* status of the traffic light in the car's vicinity. However, a location-dependent subscription as specified in Chapter 4 is valid only within a certain range. Thus, possibly, potentially interesting events concerning the traffic lights ahead of the car are not delivered to the application because the range specifications do not match at the time the notification is propagated through the network. Then, the information about the current status of the traffic light is not accessible for the client until the next change of state is published (cf. also Chapter 8). Moreover, the client does not have any control over *when* such state changes occur. A state change and its announcement as notification is bound to some event, happening outside the car application in the physical world (the traffic light changes to "green"). The question we want to discuss next is how the car application in this scenario can provide the required functionality under the given conditions.

Obviously, the application can leave the paradigm of publish/subscribe and resort to a traditional request/reply interaction. Then we have to address some additional problems: as we have no anonymous and decoupled interaction anymore, a consumer needs to find whether its current location is contained in the range of any traffic light nearby. In this case, an explicit handle is needed to contact the traffic light or its proxy for requesting the current status. Although simple, querying external sources conceptually requires an infrastructure that supports directory lookups and remote queries (see for example the "cooltown project" [Lab03]). Moreover, the client is responsible for polling the required information with all well-known consequences. Additionally, from the viewpoint of a traffic light, the tightly-coupled nature of request/reply seems inferior to the anonymous, loosely-coupled "publish" in a pub/sub system.

With a "traditional" pub/sub mechanism consumers need to update their subscriptions explicitly and are only notified if they are inside the traffic light's range at the time of the state change publication [NI97; TP00]. As a consequence, consumers are forced to wait until the next notification is published, which leads to non-negligible delays and considerable initialization latency, which might not be tolerable for mobile applications and erodes the reactivity of the pub/sub approach. An alternative is to publish a client's request for past events and route it according to existing filters to subscribers with appropriate buffers; a flexible solution, although it introduces ordering and duplication problems.

Another technique is to let traffic lights publish their status (and not their status change) with a pre-specified frequency [AFZ97]. In the worst case, applications need to wait a full update period. Hence, the choice of the frequency is a crucial parameter that affects not only applications but also resource usage. For instance, a mobile device with scarce resources, like low bandwidth wireless link and limited power supply, might suffer from heavy traffic on the wireless link when the frequency of broadcasts is too high.

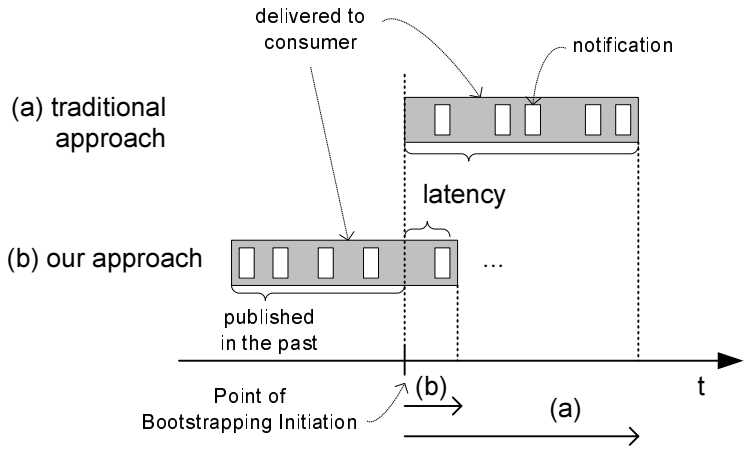


Figure 7.1: Bootstrapping latency

7.2 Subscription into the Past

The different approaches above illustrate a very simple ‘context detection’ that merely relies on the last notification as a description of current state. In general, the kind of data necessary to bootstrap an application differs widely, but in the following we concentrate on the class of applications that commence normal operation after having seen a sequence of notifications. The essential idea to diminish the bootstrap latency is to provide the consumer with a correct sequence of past notifications as if it had subscribed earlier. Figure 7.1 basically depicts a comparison of the “traditional” case (a) with our approach (b) of using past notifications in order to reduce bootstrapping latency.

The proper support by the infrastructure for mobility scenarios we propose in this chapter relies on buffering published notifications at appropriate places in the broker network and to transparently deliver them to newly subscribed consumers as required. Transparent delivery has the major advantage that it decouples clients and buffer management. Therefore, it allows for the integration of various implementation strategies for the proposed caching functionality. This includes distributed caches, proxies, peer lookup, or even centralized stores. For the client the actual reification of the required functionality is opaque. The goal of this approach is simply to define an interface between the roaming client and the buffering functionality. The concrete choice of a strategy is dependent on the actual environment and usually subject to system management choices. However, in the next section we will detail our approach of distributed caches in the broker network. Please note that the availability of an arbitrary number of past notifications cannot be assured by the infrastructure, which only mediates between applications and buffers. Consequently, in the worst case (e.g., not enough notifications were published) our approach cannot fully avoid bootstrap latency. But on the other hand, as we will show, in this case no additional overhead is added and the bootstrap latency is not increased.

By accessing information which was already delivered but is stored within the broker network applications can reach a consistent state and be set-up without listening for notification in the future. By maintaining recent published notification within the broker network the bootstrapping phase may

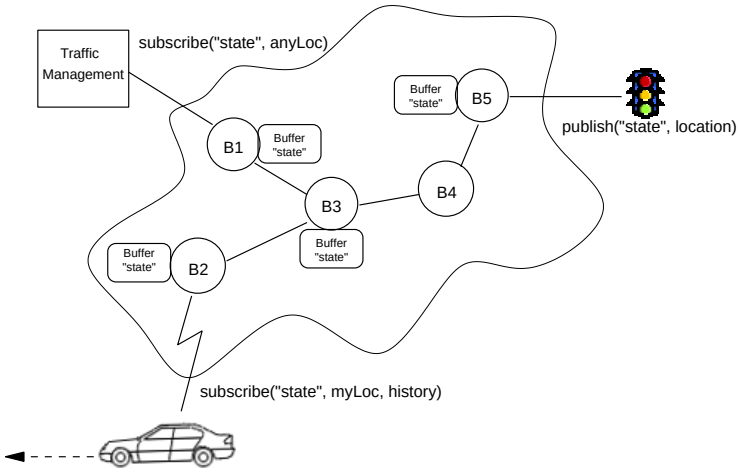


Figure 7.2: Conceptual setting: associated buffers distributed in the broker network

be sped-up significantly.

7.2.1 Basic idea

We have extended the subscription method in order to provide consumers/subscribers with the possibility to express their interest in happenings occurred in the past. This is indicated in Figure 7.2. In this case, and in addition to the subscription filter that expresses their interest, they can specify a number of *n* notifications they want to access from the past. As in the standard case, the pub/sub system delivers the last *n* notifications that match the subscription to the subscriber through the `notify` callback method as in normal operation. This makes opaque to the consumer that those received notifications have already been delivered in the past. After sending the solicited notifications stemming from the past, standard delivery of present and future notifications commence operation. Note that the system cannot guarantee that it can deliver the total number of notifications specified in the subscription. This depends on the individual policies for buffering notifications and what notifications are available in the broker network. As a consequence, the client application should not assume that the first *n* notifications in fact are part of the "past."

7.2.2 Prerequisites

In order to keep track of past notifications, buffers are implemented within the broker network and they are accessed by the different versions of the algorithm presented in the following. Conceptually, a buffer is assumed to be simply a circular log of bounded length that stores notifications matching a filter assigned to the buffer. Any broker in the network may install a history buffer as a cache of the latest notifications forwarded through this broker. In connection with a specific subscription other temporary buffers may be created as well. If we want to ensure a minimum number of notifications available for replay, border brokers have to buffer this number of notifications published by any locally attached producer. In general, any broker may maintain history buffers to improve data

placement localities, though buffers only listen to passing notifications and are empty before the first subscription initiates delivery.

Notifications travel through the broker network from the producers to the consumers along delivery paths. In the following we refer to the publishing direction as being directed *downstream*, while subscriptions and some administrative messages are directed *upstream* towards the producer. The *replay message* is of administrative kind, contains a set of notifications and is aimed downstream. Clients of the pub/sub service need not to expose any unique identifiers, but the pair (C, F) of a consumer and its issued subscription filter is presupposed to be unique; at least a unique ID can be assigned by the access broker. As introduced before in Section 4.2.1 on Page 56, filter F above is divided into an application-specific payload and the location-dependent envelope. For clarity und simplicity, we ommit this distinction here and simply use F as description of a filter.

7.2.3 Algorithm Outline

The following basic approach to subscribing with buffer replay extends the subscription process available in the REBECA pub/sub service. A subscription can now also include past notifications; the routing configuration is updated as before, but delivery of new notification is postponed; matching buffered notifications are fetched from the network; and finally, the fetched data has to be delivered before new notifications.

Issued subscriptions have to contain the number of past notifications that are to be delivered, according to the semantics given in Section 7.2.1. For the traffic light example, based on the subscriptions from the car and the traffic management control, two delivery paths through the broker network are established, as depicted in Figure 7.2. Whereby the delivery path to traffic management control is assumed to be static (hence, no number of past notifications is needed) and the wireless link between the car and broker B_2 is dynamic and ad-hoc. Here, subscriptions into the past are needed and therefore a number of past notifications is given. In Figure 7.2 denoted as $history$. As soon as the subscription $(C, F, history)$ is issued, a delivery path between producer and consumer ($\{B_5 \rightarrow B_4 \rightarrow B_3 \rightarrow B_2\}$ in the example) will be established.

In this simple example setting the inner broker B_3 plays a special role. B_3 can identify the similarity between the two subscriptions issued for the state of the traffic light. As being on the junction between the two delivery paths, once a new subscription from B_2 arrives, by applying covering and merging techniques (cf. Section 3.3 for details) or by a explicit search of the routing table, the "larger" subscription with `anyLoc` as location specification is discovered. Thus, presumably, B_3 already has a well-filled history buffer for the traffic light in question.

Immediately before a new link is activated to start delivery of passing notifications, the buffer-fetching functionality is called. The junction broker B_3 in our example selects as many of the most recent notifications from a locally kept history as necessary to meet the subscription request. These are sent as a replay message downstream the new link towards the consumer's border broker B_2. We will later suggest more advanced strategies that include history buffers at other brokers.

The border broker unpacks and delivers the replay notifications to the consumer before delivering new notifications. In general, new notifications must be delayed until the replay message is sent in order to deliver the buffered notifications first. In this simple approach, however, the replay is prepared only at the nearest branch on the delivery path, so new notifications cannot overtake the replay. However, unfortunately, the desired number of past notifications may not be available at the junction broker. Appropriate extensions to cater for this case are proposed later in this chapter.

The algorithm presented in Figure 7.3 sketches the core algorithm common to all presented ex-

tensions made in the following paragraphs. Figure 7.4 details necessary helper methods for completeness. For all extensions given below, only two methods need to be changed as it is presented in Figure 7.5. This figure particularly shows the refinements for the simplest case as outlined above.

In order to provide a framework for the most general case, the algorithm explicitly blocks delivery at the border broker in case new notifications arrive before the replay. For the basic version, as described above, this is not necessary. The timeout stops waiting for replays and starts delivering new notifications if not enough replay data was received in time; this is only necessary if multiple replays are expected.

7.2.4 Algorithm

The algorithm presented in Sect. 7.2.3 is naïve in its restriction to search only for notifications at the nearest buffer and therefore we extended it for practical relevance. One striking idea is to use more than one buffer for fulfilling a request for past notifications. But, if more buffers are considered for preparing the replay, two problems arise. First, new notifications and replays are concurrent and must be ordered correctly. And second, multiple replays may cover different producers so that reordering is not possible without identifying producers and individual notifications (a strong requirement we deliberately avoided so far). In the next subsections we suggest a number of improvements that search for more buffered notifications, cope with concurrent new notifications, join buffers of multiple producers, and reduce traffic by using sequence numbers.

7.2.4.1 Largest History Buffer

We return to the example given in Sect. 7.2.3 and specify the subscription process in more detail.

For now assume that every broker has no more than one link with a matching advertisement. In the basic approach the reduction of bootstrapping time for a consumer depends on the size of the history buffer of the first broker that is discovered on an existing delivery path. Consider for instance the network as presented in Figure 7.2 where the mobile client issues a subscription and this subscription involving past notifications depends on the broker B_3.

In order to provide a better solution the first extension includes history buffers at other upstream brokers: another broker (B_5) further upstream may have a larger buffer that could be used instead. However, new notifications may be in transit while the first broker (B_3) requests a replay of B_5's history buffer. Thus, the first broker (B_3) needs to hold notifications for the consumer, or its border broker (B_2), until a replay message is received. The replay needs to be shortened by the number of held notifications to avoid duplicating notifications. After the replay content is passed to the consumer, held notifications are delivered.

By including the contents of buffers at other brokers, potentially a larger fragment of recent history can be accessed. This assumes however, that buffers have different sizes and that it is possible to find a larger buffer in brokers upstream. In oder to provide a certain completeness, in some scenarios we even can require the border broker of a *producer* to maintain a reasonable large buffer. This way we can guarentee that a client at least can access some of the past events at all times. In this case, the reactiveness of this approach is at least as good as without buffering at all, which is the case if the subscription is the first of its kind in the system and must setup the complete delivery path between consumer and producer, and better in the average case where buffers are already set up.

```
/** upon receiving subscription (C, F) for past p events
 * via link L_N */
void receiveSub(C, F, p, L_N) {
  if (localClients.contains(C)) {
      routeTable.setHold(C, F)
      buffers.newBuffer(C, F, p) // create temp. buffer for replay
  }
  if (routeTable.includes(C, F) {
      replay(C, F, L_N, p) // prepare replay message and send
  } else {
      routeTable.add(C, F, L_N)
      propagate(C, F, p, L_N) // to all neighbor brokers with
  } // matching advertisements except L_N
}

/** upon receiving notification n from B_j */
void receiveNotif(n,B_j) {
  routeTable.route(n)
  historyBuffers.append(n)
  for(∀b ∈buffers with matching assigned F)
      b.append(n)
}

/** upon receiving replay(C, F, [n_1, ..., n_m]) */
void receiveReplay(C, F, [n_1, ..., n_m]) {
  if (buffer.exists(C, F)) { // i.e. this is a border broker
      b := buffers.get(C, F)
      b.prepend([n_1, ..., n_m])
      if(b is completely filled) {
        deliver b
        buffers.remove(C, F)
        routeTable.clearHold(C, F)
      }
  } else {
      routeTable.route(C, F, [n_1, ..., n_m]) // route replay towards consumer
  } // according to unique (C, F)
}

/** upon timeout of buffer (C, F) */
void receiveTimeOut(C, F){
   buffers.get(C, F).deliver()
   buffers.remove(C, F)
}
```

Figure 7.3: Basic algorithm for subscriptions into the past.

```
class Buffer {
    F, C, p // filter expression, consumer, past
    i, d, store // new notifications, delivered

    // ...

    void append(n) {
        if (F.matches(n)) {
            i := i + 1
            store.append(n)
            if (store.length() >= p)
                this.deliver()
        }
    }

    void prepend([n_1, ..., n_m]) {
        store.prepend([n_1, ..., n_m])
        if (store.length() >= p) {
            this.deliver()
            delivered := true }
    }

    void deliver() {
        // deliver buffer contents to C
        // ...
        delivered := true
        routeTable.clearHold(C, F)
    }

    boolean hasBeenDelivered() {
        return delivered
    }
}
```

Figure 7.4: Additional methods for the algorithm in Figure 7.3

```
void replay(C, F, L_N, p) {
    b := historyBuffers.get(C, F)
    if (b.length > 0) {
        b.sendReplay(C, F, p)
    }
}

void propagate(C, F, p, L_N) {
    for (∀n ∈ localClients\{L_N} ) {
        requestReplay(n, C, F, p)
    }
}
```

Figure 7.5: Specification and refinements of the `replay` and `propagate` methods.

7.2.4.2 Merged Histories

Looking for a broker in the delivery path with enough notifications (as it was proposed above) involves communication costs and time. Considering that during this searching time new notifications may arrive implies that fewer notifications need to be specially delivered from other brokers. This leads to the next modification of the algorithm: All queried brokers send as much of their history buffer as is available in the hope, that enough notifications were issued in the meantime to fulfill the requested number of notifications.

Coming back to the example, the first broker B_3 might decide to stop waiting for history replays and start delivering held notifications. Another broker B_4 between B_3 and B_5 might have a larger history than B_3 but not sufficiently large to satisfy the requested number of notifications. However, it could send a replay anyway increasing the probability that B_3 is able to fulfill the request based on newly received notifications plus received replays so far. B_3 needs to keep track of outstanding replay requests unless there is a timeout defined.

In order to merge replays with the local history buffer, B_3 needs to reduce the received replay by the number of notifications in its own history buffer to avoid duplicates. Since sender FIFO is guaranteed, all replays are aligned to the beginning of the first broker's (B_3) buffer. In other words, the most recent notifications are present in B_3's buffer as well as in the replay. Figure 7.6 shows the content of two brokers' history buffers, B_4 and B_3, with administrative messages and a new notification "g" is being published by the producer. At time $t+1$, B_3 receives a subscription requesting four past notifications while B_4 receives notification "g". B_3 allocates a new buffer large enough and copies the content of the history buffer to this new buffer. Next, at $t+2$, B_4 forwards "g" to B_3 while B_3 requests a replay of four notifications from B_4. Notification "g" is added to the new buffer at B_3. In $t+3$ the replay is sent to B_3. It can be seen that B_3's buffer contains the same leftmost (most recent) notifications as the replay message. The result after removing the duplicate notifications is shown at $t+4$.

With this modification, the bootstrapping delay of a consumer is guaranteed to be no larger than

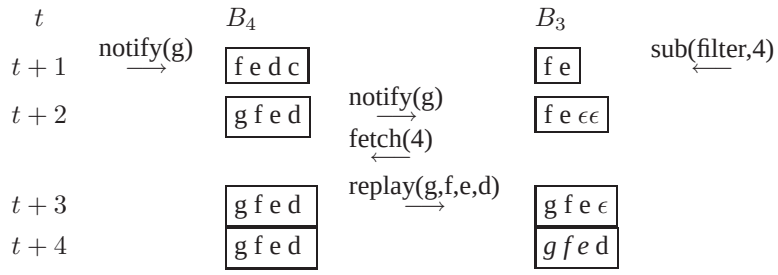


Figure 7.6: History buffers, messages and alignment of replay contents with history buffer contents

without recent history in the worst and better in the average case.

7.2.5 Further Extension for Multiple Producer Scenarios

In this subsection we present two further extensions for scenarios with more than one matching producer per client subscription. Obviously, scenarios not covered by our simple example above.

7.2.5.1 Junctions

Up to this point we have assumed a single matching advertisement per subscription, which we will relax now. *Broker IDs* help to distinguish notification sources and allow elimination of duplicates (notice that the simple alignment as depicted in Fig. 7.6 does not hold anymore).

Junction brokers are brokers that are connected to more than one peer with matching advertisements. Upon a replay request, junction brokers should query all peers in order to get a good estimation of past notifications. In general, a specific ordering cannot be assumed between replays of different producers. Replay processing may be based on first-come-first-serve, random interleaving, or timestamp ordering if logical or global-time clocks are presupposed. However, only replays from the junction broker's own history buffer will have the order of notifications the consumer would have observed if it were subscribed earlier.

When receiving replays from multiple brokers they are not guaranteed to be aligned with the local history buffer anymore thus removing duplicates is more difficult. Therefore, globally unique broker IDs must be added to a notification envelope when it is published. Since the ordering of notifications relative to the source is fixed, replays will include the same most recent notifications which allows to drop as many notifications from a replay as already present in the local buffer for this source. Notifications are always stored in the history buffer with the complete envelope.

7.2.5.2 Sequences

Although the algorithm and the extension presented above eliminates duplicates before delivering notifications to the consumer, notifications may be sent multiple times over the network since a very recent notification will be included in every replay. *Sequence numbers* can be used to shift the task to detect and eliminate duplicates from the border broker to the infrastructure.

In the same way border brokers connected to producers add their broker ID to the notification envelope, a sequence number is added. Such sequence numbers can be leveraged to eliminate du-

plicate sending. Requests for history replays include a list of tuples (b, s) with b as broker ID the producer is connected to and s as the sequence number of the oldest notification from this producer still contained in the local history buffer. As an additional benefit, the requested number of replay notifications can be reduced by the number of notifications found in the local history buffer that match the request.

7.2.6 Putting it All Together

In this section we informally describe an algorithm that takes the above said into account. Hence, three new message types are introduced for inter-broker communication: *fetchHistory(expression, list, #past notifications)*, *replay()*, and *expect(tagname)*.

Handle Subscription Message Upon receiving a subscription including past notifications p, a broker checks first if it is the responsible border broker for the issuing consumer. A buffer H with size p is created to assemble an estimation of past notifications, a counter r for outstanding requests, and a map L for lowest sequence numbers are allocated. Delivery for the requesting consumer according to this subscription is set to hold.

If the current border broker does not carry the requested subscription, the request is sent to all connected brokers that have sent a matching advertisement, hence are upstream. r is incremented for each request.

If the broker already has a matching subscription and a local history buffer matching the subscription expression exists, notifications are copied to H until the end is reached or the request is fulfilled. At the same time L is filled with lowest sequence numbers per originating broker. The number of expected notifications is decreased with each notification.

If more past notifications are needed, a *fetchHistory* message is sent to all connected brokers with matching advertisements and r is incremented for every request. If more than one broker is contacted, a junction marker is set.

Any inner broker that does not already have a matching subscription re-sends the subscription request to all upstream brokers. For each request an *expect* message is sent back downstream and a junction marker if more than one broker is contacted. In addition, an empty *replay* message is sent downstream.

If a non-border broker has a matching subscription, a local buffer H' with size p' is allocated. The broker scans the local history buffer looking for notifications that have a lower sequence number than indicated in the request. If it is the case then it copies them to H' and updates the number of outstanding notifications and the sequence number list L. All notifications in H' are included in a *replay* message and sent downstream. Whenever the number of outstanding past notifications is non-zero, corresponding *fetchHistory* messages using L' are sent to all connected brokers that have matching advertisements. *expect* messages are sent as described in the case when no matching subscription was found. Buffer H' is removed.

Handle fetchHistory Message A non-border broker handles *fetchHistory* messages similar to *subscribe* messages.
A border broker should never receive a *fetchHistory* message.

Handle Notification Upon receiving a matching *notification* for a consumer that is set to hold, a border broker copies it to the local history buffer and delivers it to all other consumers that are not

set to hold. In addition, it is queued in H and the number of expected notifications is decreased. If the number of notifications is zero, all queued notifications are delivered and all helper structures are removed. The hold marker is removed.

A non-border broker transmits the notification downstream.

Handle Replay Message Upon receiving a *replay* message, a border broker appends all included notifications to H and r is decremented. If it has been signalled before that a junction has been encountered, ordering of notifications from different producers cannot be reconstructed after this point. The border broker could try to order notifications from beyond the junction based on the notification's production timestamps.

If no replays are outstanding, all notifications from H are delivered to the consumer and the hold marker as well as all helper structures H, L, r are removed.

An inner broker relays the *replay* downstream.

Handle Expect Message Upon receiving an *expect* message, the border broker increases r. An included junction marker is stored. A non-border broker relays an *expect* message downstream.

7.2.7 Considering the Time Dimension

Subscriptions into the past can be also specified with a time bound. In this case, a subscription into the past asks for notifications that have been published *m* time units in the past (relative to subscription time). A first approximation could be achieved by synchronizing the clocks of all border brokers. In this way, at *publishing-time* border brokers attach a timestamp to notifications in order to represent the time when they have entered into the pub/sub system.

At *subscription-time* the border broker sets the time bound by simply subtracting the relative bound of the subscription from its local, synchronized time. This time reference is then used by the algorithm to search in the broker network for matching notifications with a newer timestamp.

Moreover, a combination of number of notifications and time into the past could also be useful, i.e., the last ten notifications within the last five minutes. This combination constrains the search in the buffers of the broker network. That means that there are two criteria to stop the search: (a) once the number of notifications within the reference achieves the solicited number, or (b) once a timestamp older than the reference is found.

However, we explicitly do not want to discuss problems involved by using time in distributed systems. We consider time specification as a convenient means to restrict the search space for past notifications. In the sense as time is used as part of the proposed approach it takes into account that a mobile client hardly can make any assumptions about concrete timing in changing and unknown environments. As we explicitly decouple consumer and producer in space and time, control over publishing of events solely is bound to the producer. Therefore, any expectation of a client that *time* here refers to an *exact* and global clock should be avoided.

7.2.8 Discussion

At the beginning of this chapter, in Section 7.1, we introduced the simple example of a traffic light, which changed its state to "red"just by a fraction of the second before we entered its range. There, we already characterized possible solutions to the problem of access to "historic" data. Here, we revisit and discuss them in the light of the proposed algorithm:

- *Request/reply.* We can require the producer and consumer to share a common knowledge about addresses such that we can do traditional request/reply between car and traffic light or its proxy somewhere. Besides abandoning the publish/subscribe paradigm and the additional need for an orthogonal infrastructure, we might run into the same problems regarding in-transit messages, lost messages, and duplicates as we have discussed in Section 5.5.2. There, we analyzed the shortcomings of request/reply based solutions for the relocation of mobile clients. Similar arguments against pull-based mechanisms apply here.

- *Multicast.* We can require the system to maintain location-dependent multicast groups. Although this approach provides for decoupling in space, the decoupling in time remains unsolved.

- *Frequent re-publishing.* We can require a producer of events to re-publish information frequently just in case someone is interested and just entered the range of this information. This solution is dependent on many factors and requires a sophisticated system management for tuning and optimization. It remains unclear how such an approach is applicable to producers that are not under the control of a centralized management instance. Please note that every (mobile) client also may be a producer of data interesting to other clients.

- *Flooding and client-side filtering.* We can require the infrastructure to flood information and clients to do client-side filtering. However, we discussed the drawbacks of this solution in great detail in Chapter 6. A resource-constraint device might suffer significantly. Thus, client-side filtering should be avoided.

- *Subscriptions "in-the-past".* We can require that a client adheres to a self-chosen movement graph. Then it is possible for this application to pre-subscribe to important information at future locations. However, the application itself is responsible for appropriate subscriptions to data from future locations, filtering out unnecessary information, caching informations for future locations, and unsubscribing to old locations. We proposed a similar approach in [FGHZ03], shifting the responsibilities mentioned above into the infrastructure. Conceptually seen as an extension to the approach presented in Chapter 6. However, no decoupling in time is achieved whenever a client acts "spontaneously", e.g., by *not* adhering to the chosen course of action, or by issuing new subscriptions. Then no past notifications are available and conventional publish/subscribe semantics apply.

Subscriptions "into-the-past". Finally, we can require that the infrastructure offers a means of buffering as detailed in this chapter. Besides the non-negligible complexity induced for the broker network we believe this solution to be the most flexible and complete. Assuming that clients can commence operation after having consumed a reasonably limited number of notifications for adaptation to the new environment, cache sizes can be rather small. Additionally, for pervasive environments producers can be made "aware" of the existence of client dynamics. Thus, to keep the impact on the overall system small, producers can weave "state digests" into the normal flow of information. Thereby, the number of incremental updates a potential client has to listen to is kept limited.

Even then, we cannot neglect the possibility of cache misses. As countermeasure, throughout the last sections, we devised a number of optimizations to the basic algorithm shown in Figure 7.3. We sketched several extensions trying to fulfill a client's request for past notification with growing sophistication. But, with every extension the complexity of the overall algorithm increases. Thus,

the optimal configuration of the broker network obviously is a setting where needed past notifications "always" are contained in a buffer nearby.

Unfortunately, a concrete analysis of the additional message complexity for the case of too few notifications in the nearest buffer is extremely hard to conduct. In dynamic environments the probability for such a "cache miss" depends on a number of diverse and often application-dependent factors:

- *Location-dependent subscriptions.* The class of subscription we devised the algorithm for are context-dependent subscriptions, and here especially location-dependent subscriptions. Obviously, for scenarios where clients can be assumed to subscribe to certain information, e.g., the traffic conditions in the vicinity in a car scenario, the probability that another client at the same border broker has already subscribed to the same information is very high. Thus, the needed recent notifications probably are already buffered close by. Consequently, the reactiveness is high and the probability for a cache miss low.

- *Border brokers.* In heterogeneous system settings, we have to deal with the situation that two clients at the same location access the system via different border brokers. This situation might occur when different connection technologies are in use. For instance, a client is connected via a local networking technology, e.g., WLAN, while a second client uses a GSM phone. Then, the delivery paths for those two clients might be completely different. Accordingly, the probabilities for cache misses and the mean distance to a sufficiently large buffer can vary greatly.

- *Inherent adaptive behavior.* We assume the broker network explicitly to be rich of resources. Nevertheless, the experienced quality of service is limited by the available hardware, e.g., network bandwidth and computational power. Thus, with a growing number of clients the system has to adapt its provided services. For instance, at crowded locations we have to assume a strong heterogeneity of applications in use and therefore the number of *different* subscriptions is high. The available resources have to be divided in order to accommodate to this situation. Although not explicitly detailed above in the algorithm, we assume the available resources to be distributed evenly among the subscriptions in the system. As every subscription has an associated cache, cache sizes may vary over time and broker. In a high load situation cache sizes are getting smaller in order to serve more clients. Theoretically, the probability of a cache miss for a new client at the nearest broker is higher. On the other hand, the *diversity* of subscriptions is larger, the larger the number of clients gets. Hence, the probability that another client has issued a similar subscription is larger, too. Even if a cache miss occurs, we can assume a very high probability that a broker upstream *nearby* the first broker (with a significantly lower load) has a sufficient number of past notifications available. This way, again, the additional complexity of our algorithm is reduced significantly.

Evaluation of the factors discussed above open up a wide field for experimentation and currently are covered only partially. Therefore, a complete coverage is left for future work.

7.3 Summary

This chapter is motivated by the use of publish/subscribe notification services in pervasive settings where mobility plays a prime role. In order to adapt to context changes, moving clients require an

initialization phase to commence normal operation from a valid state. Without proper countermeasures in the infrastructure the latency of a client's bootstrapping phase has the potential to severely impair the usability of the publish/subscribe paradigm in pervasive scenarios. Due to the reactive nature of event dissemination, a client has to wait until enough information is published to resume operation. The main problem we identified in this chapter is that the "settling time" of clients after startup, issuing new subscriptions, or location changes is bounded by the time a roaming client stays within reach of a single environment. Therefore, latency can become a major problem. To stress this point we introduced a simple example illustrating the case where a client (a driver assistance application in a car) must have access to certain data (the status of the traffic lights ahead) in order to operate properly. For this example we argued that the client cannot simply "stop" and wait until the next status change of a traffic light in the vicinity is observed and the car can commence operation. In this sense, the traditional publish/subscribe paradigm simply falls short. For comparison we discussed viable alternatives to publish/subscribe and identified serious weaknesses there, too.

The problems mentioned above motivate our approach, with which clients have means to access data already propagated through the broker network in the recent past. Thereby, we introduce decoupling not only in space but also in time. We establish additional buffers in the broker network of a distributed notification service. Additionally, we devised a set of possible search and consolidation strategies tailored to minimize the bootstrapping latency experienced by clients connected to the network. Matching recent notifications are looked for in buffers on the delivery paths upstream towards the producers. Unfortunately, for the clients this included a change of interface between applications and notification service. We extended the `subscribe` method by adding the possibility to specify how "far" in the past the system should go to start delivering notifications. This can be done by giving a number of recent notifications to replay for the client.

The approach proposed in this chapter seeks to bridge the divide between the algorithms devised in Chapter 5 and Chapter 6. Mobility support, as introduced in Chapter 5, decouples consumer and producer of non-context-dependent data, e.g., legacy applications, in space and time. The main purpose in pervasive environments is to bridge phases of disconnectedness and arbitrate the actual location of a client for notification delivery. Both fosters *transparency* of space and time. Chapter 6, on the other hand, uses explicit knowledge about the whereabout of clients to maximize responsiveness and minimize the bootstrapping latency for clients moving within the borders of a single border broker. The algorithm presented in this chapter combines the main characteristics of both approaches. Roaming clients, i.e., clients connecting to more than one broker over time, can have access to location-dependent context data in a transparent way. Additionally, by accessing past notification a client can be initialized faster, hence responsiveness is improved. The support of our infrastructure for both characteristics appropriately caters for the required decoupling of space and time as identified in Section 2.2.4 in Requirement 2.2.8 on Page 26.

8 A Structured Approach to the Development of Context-Aware Applications

The art of being wise is
the art of knowing what to overlook.

William James, philosopher and psychologist (1842-1910)

8.1 Introduction

In the previous chapters we have detailed the necessary extensions to a notification service as foundation for the deployment of mobile, event-driven applications in a nomadic computing environment. Thus, having those in place, we devote this chapter to a structured approach for the development of context-aware applications. The model is based on a reactive application model using finite state machines as primary abstraction.

Finite state machines perfectly model the core of context-awareness: adaptivity to changes in the surrounding. Whenever a change in the surrounding occurs that is interesting for the thread of control of a context-aware application, this can be expressed in terms of a state change in a finite state machine. In our proposed model, any change of state can be accompanied with a so-called *action*, i.e., some action is taken in order to react or respond to the detected change of context. Hence, the specification of applications is control-oriented as defined, for example, by Papadopoulos [PA98].

But, control-oriented specification of applications as finite state machines is only one side of the coin. The other side is the gap between the level of semantics an application is specified at, i.e., specification of tasks or goals to reach, and the data-driven, controller-less mode of operation of a nomadic computing system. In the nomadic computing system model, control of resources and data-flow is fully distributed and decentralized. Thus the regulation through a central controller, like a system manager, cannot be assumed. Interaction and coordination of producers and consumers of data is spontaneous, ad-hoc, and most importantly, data-centric. Due to the volatile bindings of data producers and mobile context-aware applications, the content of a data item is the distinguishing factor rather than the identity of its producer. We will discuss this issue in greater detail throughout the following Section 8.2.

Therefore, the main challenge we face in this chapter is to find a proper mechanism to transform a control-oriented, task-driven description of applications into a proper set of subscriptions for data items mediated by the distributed notification service. To develop such a mechanism is the central concern of the Section 8.3. Finally, we summarize the findings of this chapter in the concluding Section 8.4.

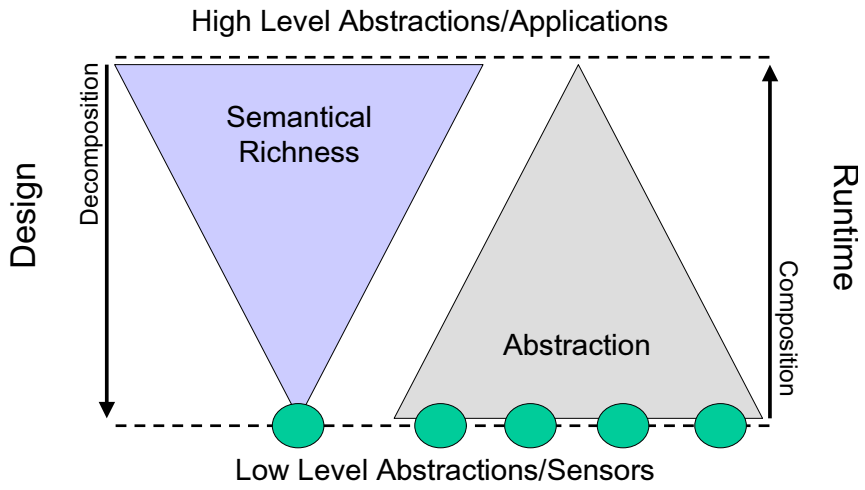


Figure 8.1: General design process

8.2 Control-driven Applications and Data-driven Environments

In theory, every single piece of context data can contribute to the execution of a context-aware application. In practice, however, information that is considered useful for the use in applications must be somehow measured or sensed in the physical world (cf. Chapter 2).

But, in a nomadic computing environment direct and exclusive control of resources in the surroundings is not within reach of a mobile context-aware application. Be it to decouple and anonymize consumers and producers, security reasons, or most likely, to make resources shareable, a roaming device cannot expect to access resources directly. It has to rely on the infrastructure for facilitating access to resources indirectly. Hence, from the application's point of view, assumptions about a concrete source of data, what syntax and semantics data at the current location has, or how and when data is acquired, are hard to make. A designer of a context-aware application thus has to resort to a more flexible approach. The specification of applications is done on a level of abstraction where reactive description of *what goal to reach* and *how to react to context changes* is favored over handling concrete instances of data directly. This is the inherent volatility we face in mobile systems. However, a significant "gap" of semantics between the producers of raw data and the applications as consumers of high-level input is introduced that must be bridged outside the mobile, context-aware application. As we argued above, the specification of coordination and interaction at designtime typically should be *control-driven*, as defined, for example, in [PA98] (see also Subsection 2.2.3.3 on Page 20). Moreover, for an application it is necessary to react to *what* happened, e.g., the presence of someone in a room where nobody should be, opposed to the *how* this fact has been acquired, e.g., by a tag-reader, an ultrasonic positioning system, or a surveillance-camera. This is necessary because of the volatile and changing environments an application might be deployed in at runtime. In this

respect, we require a reasonable separation of goal-oriented specification and actual computational concerns introduced by the concrete surroundings.

As introduced in Section 2.2.4, by definition, it is the task of an infrastructure to mediate between context producers and consumers. Usually, part of an application's context is "living" in the physical world and every state change of an observed object is reported as event in the infrastructure. Thus, interaction must be data-driven, as neither infrastructure, nor applications have control over when an event occurs in the physical world. Additionally, applications are not directly aware of data sources and vice versa; they are decoupled by the infrastructure. Both parties must rely on the infrastructure to fulfill their needs (cf. Fig. 4.1 on Page 54 in Chapter 4): a producer of events simply publishes events into the infrastructure for *multiplexing*. Multiplexing here denotes the functionality of forwarding data to interested third parties for consumption. On the other hand, a consumer specifies the generic need for input as standing request, using the infrastructure for *composition* of matching input for the application. The term *composition* hereby refers to the generation of new notifications out of one or more data items which are consumed in the process, ideally reducing the overall number of notifications in the network. But, to fulfill the need of applications for application-level input, data must be transformed into data on another semantic level, i.e., it must be interpreted to match an application's need. For example, an RFID tag held in front of a tag reader might be reported as "tag reader event." The infrastructure is doing the necessary *multiplexing* of the event, i.e., it forwards it towards interested third parties according to the notification's content and stated interests. Here, interaction is data-driven. The same content might then be used within a "smart conference room" application to determine who is in the room, as well as for a health monitoring application of one of the participants. The original "tag-reader event" might have undergone interpretations to map it onto an "id number" for the conference room application and a "patient id" for the health monitoring application, accordingly.

However, at the junction of mobile devices roaming in the physical world and the infrastructure fostering coordination for applications and distribution of context, we have to find a model of context-handling that is bridging the "gap" between the data-driven nature of data acquisition and the control-driven specification of context-dependent applications. At the same time it must also be suitable for implementation within the infrastructure (cf. Section 8.3).

The next section will introduce a formal model which takes into account the capabilities and characteristics of a notification service in order to build context-dependent applications. We will then show how this model can be applied to the design of context-aware applications by taking a strongly simplified example from the domain of health care, where we started to explore this approach [Rei02].

8.3 A Model of Context and Context-Handling in Notification Services

Throughout the last section, we have motivated the need for a model of context-handling, bridging between control-driven specification of applications and the data-driven nature of the underlying infrastructure using a notification service as an efficient intermediary. The downside of this choice is that it does not come without costs. We have to tailor the model for handling external context in a way it is implementable by means of a distributed notification service. Therefore, in Section 8.3.2 and 8.3.3 two abstractions are introduced, which are directly deployable in a notification service. The first abstraction we introduce is *Parameter*, which is classifying *event notifications* according

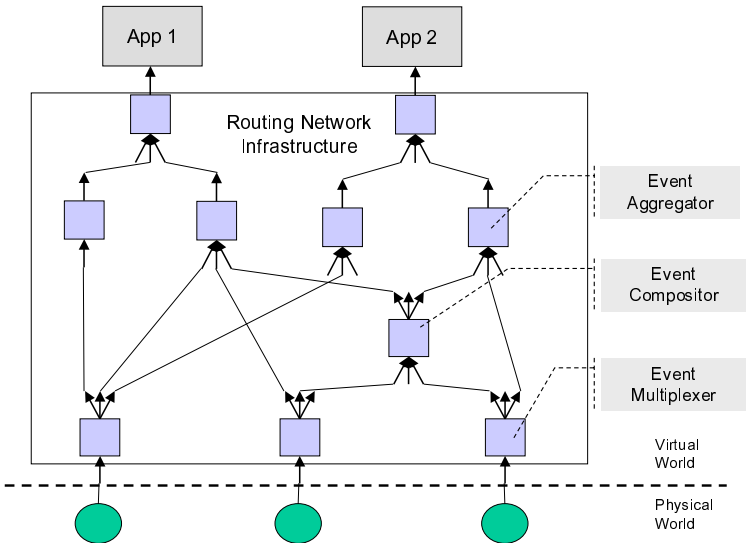


Figure 8.2: Logical view on coordination within REBECA

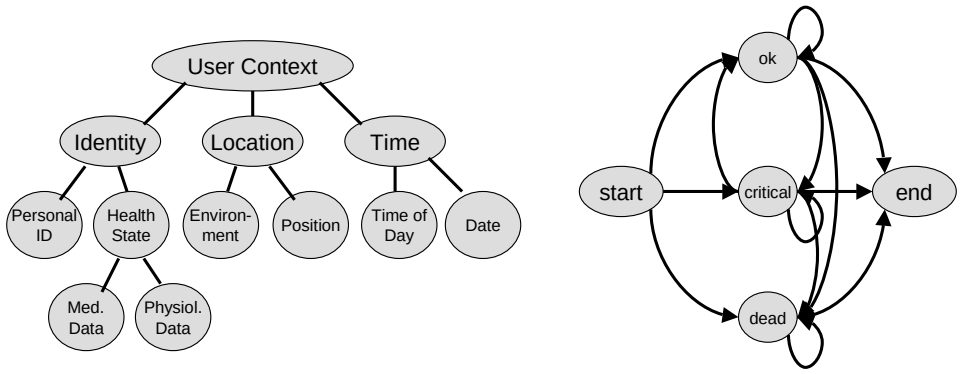
to the raw data they contain. Parameters thereby constitute the most basic building block for the generation of context information. The second abstraction is *Interpretation*, a means with which new data (and hence new notifications, cf. Sect. 8.3.3) is generated by consuming and interpreting some input data.

The intended use within our model is to build up hierarchies of interpretations (cf. Fig. 8.2), thereby generating high-level context in a finite number of steps from low-level parameter data. The abstractions used within the notification service to implement interpretations are *event composition* and *event aggregation*.

The whole model of context-handling is constructed around those building blocks to keep it applicable together with a notification service. The result is an overall design process for context specification and subsequently context-handling as shown in Fig. 8.1 and detailed throughout the next subsections:

Application design. At designtime we require an application to be specified in terms of semantic meaning, i.e., by specifying *how to react to context changes*. For the specification of reactive systems, design paradigms for control-oriented behavior, e.g., *statecharts* [HG97; HN96] and the *state pattern* [GHJV95], were introduced. Taking this as a starting point, in Section 8.3.4 and Section 8.3.5, we show how at designtime applications can be decomposed along the lines of parameter and interpretation abstractions.

Runtime. At runtime, an application is deployed using the primitives offered by using a notification service. Each “request” for parameter data can directly be expressed by a corresponding



a. Relevant context (example) b. FSM_{health} state transitions (simplified)

Figure 8.3: Relevant context for the MobiHealth application (a) and FSM_{Health} state transitions (b)

subscription issued, which matches data published by producers. *Event compositors* take the role of interpretations. Their role at runtime is twofold: (i) they subscribe to data matching their input specification. This can be parameters, but also output of other event compositors (cf. Fig. 8.2), thereby building up the data-flow eventually providing context to applications; and (ii) by typically consuming more notifications than they produce, they provide means for external filtering and aggregation of data, preventing a resource limited device from being flooded by too much data to filter and process locally. An especially important aspect for mobile devices or appliances as we discussed in Section 2.2.4.3 and formulated in Requirement 2.2.10 on Page 27.

8.3.1 Introducing a Running Example

Throughout the following sections, we want to illustrate our model by modeling an extremely simplified version of a *Mobile Personal Health Monitor* (MobiHealth) as it was originally designed and implemented using a first version of the model as presented in [Rei02]. The purpose of the example system is to monitor the health state of a mobile user and to set-off an automated emergency alarm whenever some crucial health state becomes critical. The original target group of users were elderly people with cardiac problems, where fainting, accompanied by a drop in blood pressure, might indicate a critical situation, triggering some emergency measures.

Nonetheless, the model we introduce is not restricted to this application, but is applicable to other application domains and examples as well.

For the sake of our example, the goal we eventually want to achieve at designtime is to identify data items corresponding to the context hierarchy as shown in Figure 8.3(a) and map them to context data which is then used to trigger state changes within the application as it is shown in Figure 8.3(b) at runtime.

Obviously, raw data, such as location or environmental temperature, is provided externally by sensors embedded into the surrounding physical space. We show how this data can be stepwise transformed into specific context information on a higher semantic level. Such context information

then is used to determine state changes and appropriate actions on the application level.

In the following sections, at the same time we introduce the formal model and show its application by relating it to our example of the personal health monitoring system.

8.3.2 Parameters

When modeling context, the most basic input coming into the system is *raw data*, usually produced by hardware sensors and corresponding driver software. In order to move away from the details of dealing with different hardware platforms, software frameworks like the ContextToolkit [SDA99] are put in place to "wrap" low-level details and foster re-usability and flexibility. Instead of programming against a particular hardware controller, software is developed, using a unified layer of interfaces and well-defined data formats. For example, temperature sensor A measures environmental temperature in "degrees Centigrade," whereas temperature sensor B measures the same data using "degrees Fahrenheit." An appropriate wrapper architecture can make both sensors conform to a well-known data definition `Temperature`. Usually, for data obtained by measurement in the physical space, a physical unit is used, like in the example above "degrees Centigrade."

Multiplexing. Instances of parameters are injected into the system by producers of data, e.g., a sensor periodically sending the current temperature reading. Any producer publishes data as a sequence of notifications via its associated border broker. The broker forwards the notifications towards interested third parties, like our context-aware application under consideration, according to a table of subscriptions matching the content of the notification. This way, data is propagated through the network of event brokers towards consumers. As the notification is duplicated every time data is forwarded in more than one direction at a time, the routing network serves its function as a multiplexer of data.

Within our formal model, all available parameters constitute the global set $\mathbb{P}$. A parameter serves as the smallest building block of context-dependent systems. Goal of the design process is to identify the subset $\mathbb{P}_{App}$ of parameters used within a context-aware application App. Deployed at runtime, parameters can be expressed as subscriptions in the notification service.

For practical reasons, we distinguish between *single-valued* and *multi-valued* parameters. For each single-valued parameter a domain and a unit are defined.

Thus, a single-valued parameter p_{sv} is defined as a triple, with a domain $\mathbb{D}$, a value v, and a unit u as its elements:

$$p_{sv} = (\mathbb{D}, v, u), v \in \mathbb{D}$$

A multi-valued parameter p_{mv} is defined as a set of k single-valued parameters p_{sv}:

$$p_{mv} = \bigcup_{0 \leq j < k} \{p_{sv_j}\},\ k \geq 1$$

Obviously, the domain of p_{mv} is the cross-product of its elements' domains:

$$\mathbb{D}_{p_{mv}} = \mathbb{D}_{p_{sv_0}} \times \ldots \times \mathbb{D}_{p_{sv_{k-1}}}$$

Consequently, we get a set of single-valued and a set of multi-valued parameters, $\mathbb{P}_{sv}$ and $\mathbb{P}_{mv}$, respectively, with $\mathbb{P} = \mathbb{P}_{sv} \cup \mathbb{P}_{mv}$.

In the course of this chapter, we assume a well-defined data acquisition framework to be in place and consequently the set of possible parameters that can be used in an application to be finite. The

namespace used within this acquisition framework is standardized. For the underlying system model of nomadic computing it is also reasonable to assume that a core set of parameters is available at each location visited. Especially, we assume location and location detection to be available.

In order to structure the given parameters such that we will eventually be able to use instances of parameters (*raw data*) to compose context, all parameters are logically grouped into l subsets. The grouping criterion depends on application semantics. The sets are non-empty and pairwise disjunctive. Hence, let $\mathcal{P}$ be a partition, i.e., a set of l partition elements:

$$\mathcal{P} = \{\mathbb{P}_0, \ldots, \mathbb{P}_{l-1}\},\ \mathbb{P}_i \in 2^{\mathbb{P}} \setminus \emptyset,$$

$$\text{with } \forall\, p \in \mathbb{P}.\, \exists\, \mathbb{P}_i \in \mathcal{P} :\ p \in \mathbb{P}_i,\ 0 \leq i < l$$

$$\text{and } \mathbb{P} = \bigcup_{0 \leq i < l} \mathbb{P}_i,\ \mathbb{P}_i \in \mathcal{P},\ \ \mathbb{P}_i \cap \mathbb{P}_j = \emptyset,\ \forall\, 0 \leq i, j < l,\ i \neq j.$$

The restriction to pairwise disjunctive sets is not mandatory but for structuring and simplifying the overall design.

On these sets we define a surjective grouping function: $g : \mathbb{P} \rightarrow \mathcal{P}$, which maps each parameter to a partition element. The domain of a partition element $\mathbb{P}_i$ is the cross-product of parameter domains:

$$\mathbb{D}_{\mathbb{P}_i} = \mathbb{D}_{p_{i,0}} \times \mathbb{D}_{p_{i,1}} \times \ldots \mathbb{D}_{p_{i,|\mathbb{P}_i|-1}},$$

where the p_i are single or multi-valued parameters.

Example. For our running example, we identify three main context categories (cf. Fig. 8.3(a)): first, *identity*, with ID representing a user, his medical history MedData, his blood group BG, and physiologic parameters (for health state determination) such as heart rate HR, blood pressure BP and skin temperature ST. Second, *location and environment*, with geographical position POS, light intensity LI, temperature T and barometric pressure AP. And the last category is *date and time*.

Hence, we have a "global" set of parameters we can use. For the example, we define:

$$\mathbb{P} = \{\mathsf{HR},\ \mathsf{BP},\ \mathsf{ST},\ \mathsf{POS},\ \mathsf{LI},\ \mathsf{T},\ \mathsf{AP}\}$$

Optional partitioning $\mathbb{P}$ gives us the following partition:

$$\mathcal{P} = \{\mathbb{P}_{PHYSIO},\ \mathbb{P}_{ENV},\ \mathbb{P}_{LOC}\}$$

with the partition elements (obtained by applying the mapping g):

$$\mathbb{P}_{PHYSIO} = \{\mathsf{HR},\ \mathsf{BP},\ \mathsf{ST}\},\ \mathbb{P}_{ENV} = \{\mathsf{LI},\ \mathsf{T},\ \mathsf{AP}\} \text{ and } \mathbb{P}_{LOC} = \{\mathsf{POS}\}.$$

Whereby $\mathbb{P}_{ENV}$ and $\mathbb{P}_{LOC}$ obviously have to be acquired externally.

For the demonstration of parameter handling we take the blood pressure parameter BP as an example. BP is a multi-valued parameter. It consists of the single-valued parameters *systolic*, *diastolic* and *mean aortic blood pressure*:

$$\mathsf{BP} = \{\mathsf{BP_{sys}},\ \mathsf{BP_{dias}},\ \mathsf{BP_{map}}\}$$

The unit of all three pressure types is `[mmHg]`. For the single-valued parameters we may choose the following domains:

$$\mathbb{D}_{\mathsf{BP_{sys}}} = \{0, \ldots,\ 300\},\ \mathbb{D}_{\mathsf{BP_{dias}}} = \{0, \ldots,\ 200\},\ \text{and } \mathbb{D}_{\mathsf{BP_{map}}} = \{0, \ldots,\ 300\}.$$

Some "normal" values of $\mathsf{BP}_{\mathsf{sys}}$, $\mathsf{BP}_{\mathsf{dias}}$ and $\mathsf{BP}_{\mathsf{map}}$ are 120 mmHg, 80 mmHg and 92 mmHg, respectively. BP's domain is defined as the cross product of its elements' domains:

$$\mathbb{D}_{\mathsf{BP}} = \mathbb{D}_{\mathsf{BP}_{\mathsf{sys}}} \times \mathbb{D}_{\mathsf{BP}_{\mathsf{dias}}} \times \mathbb{D}_{\mathsf{BP}_{\mathsf{map}}}.$$

8.3.3 Interpretation

The next step is to introduce the notion of an *interpretation*. Interpretations operate at the junction between the design process of an application and its actual deployment at runtime. Their use therefore is twofold: on the one hand they are used as abstraction from an actual data source as they describe context on a semantic level; on the other hand they serve as a convenient means for the infrastructure to preprocess and filter data, thereby reducing the number of notification to be processed by mobile clients. Thus, goal of the interpretation process is to build a well-defined interpretation hierarchy, deployed in the infrastructure, which generates high-level application context out of a potentially large number of raw data items, floating through the system.

Before diving into details, take a simple example from a different application domain: let us assume a program has to decide what icon to display to lab students sitting in a windowless lab in order to indicate the weather conditions outside a building (e.g., comparable to the *kweather* applet of the K Desktop Environment [Env03]). It displays a friendly sun, if temperature, time, and sunlight intensity show that it is "a sunny but not too hot day". On the other hand, on a really hot day, a different icon is shown (e.g., a sweating sun). Goal must be, to construct a hierarchy of interpretations, eventually, converting and concentrating low-level data, e.g., (`((temperature>100)&&(unit==''F''))`, into context, meaningful for reaching the goal of displaying the appropriate icon ("really hot").

For our model, by design, we require the interpretation process to be goal-oriented *and* data-centric at the same time, i.e., every interpretation is supposed to generate data "closer" to the goal of input data to control-driven applications *and* to be compatible with the underlying asynchronous data-driven paradigm of publish/subscribe.

To achieve this, two obvious complementary "conditions" must hold:

1. *Decomposition.* At designtime, any application-level context which is subject to interpretations in the infrastructure can eventually be described in terms of *parameters* and the decomposition process, as introduced in this section, is applicable.

2. *Composition.* Any application-level context information must be composable at runtime in a finite number of steps out of actual instances of *parameters*.

The first item simply states that the process of decomposition as specified in this section results in a well-defined set of *parameters* $\mathbb{P}$. Complementary, when deployed at runtime, any interpretation hierarchy eventually will produce the intended context information for some application.

Formally, the process of interpretation is based on an interpretation mapping i, which is a mapping of $m \leq |\mathbb{P}|$ parameters and $n \leq |\mathbb{I}| - 1$ results of interpretations to a new interpretation, the latter being on a higher level of abstraction, whereby $\mathbb{I}$ holds the set of all interpretation mappings available:

$$\begin{array}{lcl}
i: & \overbrace{\mathbb{D}_{p_0} \times \ldots \times \mathbb{D}_{p_{m-1}} \times \Gamma_0 \times \ldots \times \Gamma_{n-1}}^{\mathbb{D}_i} & \rightarrow \Gamma_i,\ 1 \leq m \wedge 1 \leq n \ \text{ or} \\
i: & \mathbb{D}_{p_0} \times \ldots \times \mathbb{D}_{p_{m-1}} & \rightarrow \Gamma_i,\ 1 \leq m \wedge n = 0 \ \text{ or} \\
i: & \Gamma_0 \times \ldots \times \Gamma_{n-1} & \rightarrow \Gamma_i,\ m = 0 \wedge 1 \leq n
\end{array}$$

In the definition above, $\mathbb{D}_{p_j}$ $(0 \leq j < m)$ are the domains of parameters $p_j \in \mathbb{P}$ and $\Gamma_i = \{d_{0,i}, \ldots, d_{|\Gamma_i|-1,i}\}$ contains all possible notifications that can be created by i. The input Γ's are defined analogous to Γ_i. interpretation mappings (from now on called *interpretations*) whose domains only consist of parameter domains (see the second definition of i above) are called *first-level interpretations*. If the ranges and domains of interpretations are compatible, compositions of interpretations are allowed and an *interpretation hierarchy* can be built. When moving upwards in that hierarchy, each hierarchy level provides a higher level of abstraction. Obviously, by this definition we require interpretations to remain stateless in order to make them usable for more than one single application.

One possible implementation of interpretations is using a notification service together with *event composition* and *aggregation*. Without going into the details, common event compositors include *conjunction, disjunction, sequence* or *negation*. Approaches to implement event composition in event-based systems are described, e.g., in [LCB99; CBB03; PSB03]. At runtime event compositors are deployed within the network of event routers and are specialized to implement a specific composition, i.e., they consist of notification filters, matching on predefined data templates in the *content* of a notification. Depending on the type of composition one or more events are needed to satisfy a specified composition criteria. Opposed to "conventional" notification filters, where a positive match only serves the purpose to decide in which directions a notification must be delivered to reach potential consumers, a compositor consumes events in order to produce an appropriate event of higher semantic meaning instead, which is then routed, accordingly. Thereby, typically, the number of events in the system is reduced. For the underlying notification service notifications generated by compositors are not handled differently to conventional notifications, thereby not weakening the data-driven paradigm of event systems. Thus, semantic meaning must be implemented elsewhere, i.e., notifications produced by one compositor is used as input to other compositors, which subscribe to notifications of the generated kind. Hereby, a self-organizing *hierarchy* of interpretations is built on top of a notification service, eventually generating context as input for an application by using data-driven sources.

Example. After designing the intended behavior of some application `App`, we get a description of `App` like the one shown in Fig. 8.3b. The finite state machine shown describes the reactive behavior, i.e., state changes, on the event of some new context.

For the sake of simplicity, the only top-level interpretation in this example is i_{Health}, which is defined as:

$$i_{Health} : \underbrace{\mathbb{D}_{\mathbb{P}_{PHYSIO}} \times \mathbb{D}_{\mathbb{P}_{ENV}} \times \Gamma_{i_{Location}}}_{\mathbb{D}_{i_{Health}}} \rightarrow \underbrace{\left\{ \begin{array}{l} \mathsf{ok}, \mathsf{critical}, \\ \mathsf{dead}, \mathsf{unknown} \end{array} \right\}}_{\Gamma_{i_{Health}}}$$

However, it is worthy to note that the interpretation of the user's position as part of i_{Health} (represented by $\Gamma_{i_{Location}}$) is done by a separate (second-level) interpretation $i_{Location}$, specialized on location detection. Thereby specific knowledge about the health state is separated cleanly from specific knowledge about interpretation of location information.

Finally, in a recursive process, the set of necessary parameters to concretize such interpretation easily can be obtained and expressed in a set of subscriptions.

The design process (cf. Sect. 8.3.4) results in sets for parameters, their partition, and interpretations:

$$\mathbb{P}_{\mathsf{MobiHealth}} = \mathbb{P}_{PHYSIO} \cup \mathbb{P}_{ENV} \cup \mathbb{P}_{LOC} \text{ and } \mathbb{I}_{\mathsf{MobiHealth}} = \{i_{Health},\ i_{Location}\}$$

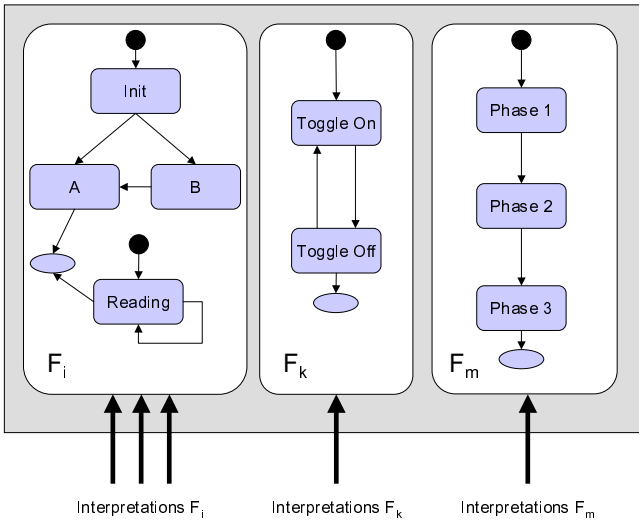


Figure 8.4: Statecharts and external interpretations

8.3.4 General Scheme for the Development of Context-aware Applications

Given the definitions of interpretations and parameters it is possible to build applications using external context. In this section we show the overall design process of an application App, before we explain the underlying model of finite state machines in more detail. Goal of the process shown here is to recursively design well-defined hierarchies of event compositors, event aggregators and multiplexers (cf. Fig. 8.2), having an application-centric semantics at the top and a data-centric realization at the bottom of any such hierarchy, describing the distribution of control during design. At runtime, the same hierarchy describes the flow of data, needed to generate the context for an application.

- The first step is to define the goal of an application in terms of reactive behavior to context changes. As starting point, we assume the proceeding detailed in the next section to be applied and eventually as result a top-level representation of App as shown in Fig. 8.4.
- Once goal and behavior are specified, it must be determined, which top-level interpretations, i.e., what context information, is needed as input for application App and its separate secondary finite state machines. For every transition defined, an action a is added to the set of actions to define a reactive behavior where needed.
- The heart of the shown process is to *recursively* concretize the top-level *interpretations* (Section 8.3.3) into *parameters* (Section 8.3.2). The recursion basically consists of four steps:
  1. For each interpretation i identified, add i to the set of contributing interpretations.
  2. Check the set of input parameters for each interpretation i. Add all new parameters of i to the set of parameters $\mathbb{P}_{App}$ contributing to App.

3. Check the set of lower-level interpretation i' that are used as input for the current interpretation i. Add them to the set of interpretations $\mathbb{I}_{App}$ contributing to application App.
4. For each interpretation i' newly added, repeat the recursion until no new interpretation is identified and only parameters were added in step 2 and 3.

In the next section we show that the result of the process presented here can be expressed as a well-defined finite state machine App, which adheres to a control-oriented semantics on the one hand and is directly deployable in a data-driven mobile environment using a notification service, like the one specified in this thesis.

8.3.5 Modeling Context-handling with Finite State Machines

This section is dedicated to the details of the design process presented in the previous section. Several models for task-oriented design were proposed in the literature, probably the best known are *state patterns* [GHJV95] and *statecharts* [HN96]. Both models have in common that the actual design of applications is based on the definition of finite state machines. Interestingly, in [HG97] Harel and Gery show, how statecharts can be applied to object-oriented modeling under the assumption of a central controller, which is partly comparable to the problem we tackle in this chapter: coordination of distributed, independent and anonymous components to reach a common goal. We then show the appropriateness of a similar approach for building mobile context-aware applications. Following their idea, we require applications to be specified as shown in Fig. 8.4: as basic building blocks, an application can be composed out of several independent "processes" or modules. Thereby, independent parts can be modeled separately from each other as they are implicitly concurrent. On the other hand, within each module execution serves a common goal and shares a common thread of execution, which can be modeled conventionally by using states and state transition, i.e., finite state machines.

For modeling external context as input, the general idea is to *explicitly group the application around the top-level interpretations used*. Then, we use this initial set of interpretations as starting point for the refinement process introduced in the previous section. For each module being part of the application, by design, the internal thread of control is dependent on the interpretations used as external input at some point. Hence, every output value of interpretation i in the result set Γ_i corresponds to a separate state transition in the finite state machine representing this module. Associated to each transition is a set of *actions*, defining the reaction to the change of context indicated through i. Thereby, we design the context-aware part of application `App` as part of the finite state machine for an application.

Formally, the above approach has two implications: (i) at top level, we can model applications as a single finite state machine consisting of a *set* of secondary finite state machines, each corresponding to an independent part of the application; and (ii) each secondary finite state machine correlates to the interpretations and parameters contributing to the input for the progress of this particular finite state machine.

Thus, we start by giving a definition for the top-level finite state machine `App` as result of the design process. Let $\mathbb{P}_{App} \subseteq \mathbb{P}$ and $\mathbb{I}_{App} \subseteq \mathbb{I}$ be the sets of parameters and interpretations relevant as input for an application `App`. On the application level, parameters and interpretations are used to take application-dependent actions, which we denote as $\mathcal{A} = \{a_0, a_1, \ldots\}$, for easier use. For example, user interaction as result of some context change now can be modeled by defining some action a_i as

part of $\mathcal{A}$. Also, the need to reinitialize the application whenever a location change occurs can be modeled as an application specific action.

Further, let $\mathcal{FSM}$ be the set of all finite state machines. Every application App defines a subset FSM_{App} of $\mathcal{FSM}$. Hence, we define App as a 5-tuple:

$$App = (\mathbb{P}_{App},\ \mathbb{I}_{App},\ FSM_{App},\ s_{global},\ \mathcal{A}),\ FSM_{App} \subseteq \mathcal{FSM},$$

where s_{global} is a distinct state, the so-called *global state*, a representation of the global state an application currently is in when all concurrent state machines constituting App are considered.

Next, in order to make the correspondence between interpretations and state changes in concurrent modules of the application App more concrete, we have to define a *Moore automaton* for each interpretation $i \in \mathbb{I}_{App}$, first. $F_i \in FSM_{App}$ is a 7-tuple:

$$F_i = (Q, \Sigma, \Delta, \delta, \lambda, q_{start}, E) \text{ with } \delta : Q \times \Sigma \rightarrow Q,\ \lambda : Q \rightarrow \Delta,\ E \subseteq Q,$$

where Q is a set of states, Σ defines the input and Δ the output of F_i, δ is a state transition function, λ an output function, q_{start} a dedicated initial state, and E is a set of final states, defining a subset of Q. Within our model we make the distinction between *core states* and *bracket states* of F_i. *Core states* are states that have a corresponding interpretation in i (to be more accurate: they have a corresponding value within the result set of i, Γ_i, which are created by a surjective mapping). As not every state change correlates to external context changes, we define *bracket states* to be states, which are not related to any interpretation i. In order to adhere to the semantics of the statechart model, we define q_{start} and q_{end} (whereby always $q_{end} \in E$) to be distinct bracket states that are part of *every* finite state machine in $\mathcal{FSM}$.

Together, we get:

$$Q = Q_{core} \cup Q_{brackets},\ \ \{q_{start}, q_{end}\} \subseteq Q_{brackets} \text{ and } q_{end} \in E.$$

The state transition are covered by the *transition function* δ. Here, we have the strongest semantic ties between the context needed in an application and the data acquired outside the application. As introduced above, δ is mainly based on the interpretation i used. We distinguish:

1. *Autonomous transitions.* Autonomous transitions occur wherever an interpretation *directly* causes a state change in F_i. Only the actual result of i is used for determining when to change states, i.e., F_i's input alphabet Σ equals the result set of i: $\Sigma := \Gamma_i \cup \{\text{end}\}$. For example, whenever an observed physical object changes state, a corresponding virtual counterpart should change its state, too. In Fig. 8.4 the `Toggle On/Off`-state is a good example for an autonomous state change.

2. *Non-autonomous transitions. Non-autonomous transitions* are used whenever an interpretation is not the only base for the decision when to change from one state to another. The application "consults" an interpretation i, but the decision of whether to change states is done "elsewhere", e.g., by displaying the content of the interpretation to the user for the final decision what to do. The interpretation then acts as information, but the decision is internal to the application. Formally, Σ then must be written as $\Sigma := \Gamma_i \cup \Sigma_{App}$, where Σ_{App} is an application-dependent input, representing the internal decision process.

Autonomous transitions are most desirable for a clean separation of coordination and computational concerns. But, in mobile applications many examples can be found where other factors play an

important role and forbid automated state changes. For example, user feedback for decision making (e.g., "click to buy") or locally available status information, like low battery life, might influence a state change or have a higher priority than external context.

Next, the global state s_{global} is a $|\mathbb{I}_{App}|$-tuple of all states the separate and completely concurrent finite state machines are currently in:

$$s_{global} = (q_{current,0}, \ldots, q_{current,|\mathbb{I}_{App}|-1}),\ q_{current,j} \in Q_j,$$

with Q_j ($0 \leq j \leq |\mathbb{I}_{App}| - 1$) being the set of states of finite state machine $F_j \in FSM_{App}$. Based on the global state s_{global}, the mapping γ is defined as:

$$\gamma : Q_0 \times \ldots \times Q_{|\mathbb{I}_{App}|-1} \rightarrow 2^{\mathcal{A}} \setminus \emptyset$$

The function of γ is that of a selector. Because of concurrency, whenever a subordinary finite state machine F_i of `App` changes state, an associated *action* $a_{i,j}$ can be executed in reaction to this state change (cf. also [HN96] for a discussion on *event-condition-action-rules* in this context). In our model γ has to select the appropriate action relative to a given change of state. For the sake of simplicity we assume all actions to be relative to each subordinated finite state machine and therefore independent of the state of other parts of the application `App`. The advantage is that then s_{global} is a "virtual" state, i.e., does not have to be actually determined and, more importantly, the selection of an action a is determined exclusively by the finite state machine the state change occurred in.

Example. For our example, the set of associated actions to be taken by the application MobiHealth is as follows:

$$\begin{aligned}\mathcal{A} = \{ &\ \text{Do nothing},\ \text{Send } \textit{health state} \text{ and } \textit{position} \text{ information},\\ &\ \text{Communicate with user},\ \text{Initiate rescue operation}\ \}\end{aligned}$$

Since we have i_{health} as only input for the application, the definition of a single finite state machine as "module" is sufficient: $FSM_{\mathsf{MobiHealth}} = \{F_{Health}\}$. F_{Health} is defined as

$$\begin{aligned} F_{Health} &= (Q, \Sigma, \Delta, \delta, \lambda, q_{start}, E) \text{ with}\\ Q &= Q_{core} \cup Q_{brackets},\ Q_{core} = \{q_{ok},\ q_{critical},\ q_{dead},\ q_{unknown}\},\\ E &= \{q_{end}\},\ \Sigma = \Gamma_{i_{Health}} \cup \{\mathsf{end}\}. \end{aligned}$$

The states of Q_{core} are directly derived from $\Gamma_{i_{Health}}$. The state transition function δ then is defined as:

$$\begin{aligned} \delta(q_i, j) &= q_j, && \text{for } j \in \Sigma, q_i \in Q_{core} \setminus \{q_{dead}\}\\ & && q_j \in Q_{core} \cup \{q_{end}\}\\ \delta(q_{dead}, \mathsf{end}) &= q_{end} &&\\ \delta(q_{dead}, \mathsf{dead}) &= q_{dead} &&\\ \delta(q_{start}, j) &= q_j, && \text{for } j \in \Sigma \setminus \{\mathsf{end}\},\ q_j \in Q_{core} \end{aligned}$$

The first of δ's equations shows that remaining in a state is allowed (that is, if $i = j$). For the example application, we do not define an output function λ. Figure 8.3b. shows some state transitions of F_{Health}. To keep this example simple, we left out the $q_{unknown}$ state, which is semantically more problematic and adds nothing to this example. The function γ determines how the application has

to react in different states, whereby here only core states are of interest. The reaction to some state change is defined by the function γ:

$$\begin{aligned}
\gamma(q_{ok}) &= \{\{\text{Do nothing}\}\} \\
\gamma(q_{critical}) &= \{\{\text{Communicate with user}\}, \{\text{Initiate rescue operation,} \\
&\quad \text{Send } \textit{health state} \text{ and } \textit{position} \text{ information}\}\} \\
\gamma(q_{dead}) &= \{\{\text{Send } \textit{health state} \text{ and } \textit{position} \text{ information}\}\} \\
\gamma(q_{unknown}) &= \{\{\text{Communicate with user}\}\}
\end{aligned}$$

In state q_{ok} no action must be taken because the status is normal. In $q_{critical}$, we have detected that something is outside the semantic specification[1]. The obvious reaction to the transition into $q_{critical}$ is, to try and get feedback from the user[2]. Depending on the (lack of) feedback from a user, the reaction and the next transition might be independent from the interpretation function, even if the system is reporting that the state changed back to normal. If the user does not react, all relevant data is sent to a base station and a rescue operation must be initiated immediately. Obviously, the state $q_{unknown}$ should never be reached and serves as default state for malfunctions.

8.4 Summary

A significant class of applications in mobile computing are context-dependent applications, like location-based services, which operate in a volatile execution environment. Control-based coordination promises to be a convenient abstraction for the specification of applications which have to adapt to context changes, as it corresponds to prevailing imperative programming languages. On the other hand, application context is based on external data sources available in the infrastructure, which usually are not part of the same application domain. For example, in the domain of health monitoring the state of a patient can be dependent on the location (inside or outside) which is detected by external location sensors. Moreover, mobile systems must be built loosely coupled and therefore, data often is generated independently from a concrete application, inherently following a data driven paradigm. In this paper we present a scheme to decompose control driven context-dependent applications into smaller components, which, eventually, are based on actual entities available in the environment. Finite state machines are used to model the application and they are systematically refined to match data available in the infrastructure. To connect data-driven sources and the task-oriented applications a notification service is used. We illustrated our approach with an example taken from the health-care domain.

[1] The state transition from q_{ok} to $q_{critical}$ is a good example for an *autonomous transitions* as defined in Section 8.3.5. The change is solely based on external input.

[2] Complementary, a good example for a *non-autonomous transition*.

9 Implementation

I mean, if 10 years from now, when you are doing something quick and dirty,
you suddenly visualize that I am looking over your shoulders
and say to yourself, Dijkstra would not have liked this,
well that would be enough immortality for me.

We humbly disagree: 10 years of Dijkstra is just not long enough;
may he happily haunt our consciousness for 1010 years.
Such an increase is more befitting his stature.

From: More Java Pitfalls[DSAR03]; dedication to Edsger W. Dijkstra

9.1 Introduction

This chapter is dedicated to the implementations done in the context of this thesis. We focus the following description on two main aspects: first, the integration of support for mobile clients into the underlying distributed notification service REBECA and second, the extensions made to facilitate location-dependent subscriptions as an integral part of the routing infrastructure. Although substantial additions were made, we show in this chapter that the design of the algorithms introduced in earlier chapters are beneficial for the implementation effort: either the changes necessary can be masked in such a way that the core REBECA system is not aware of them, or they are completely orthogonal to the existing functionality. Therefore, the core functionality of REBECA is not impaired.

The outline of this chapter is as follows: first, in Section 9.2 we give implementation details on the distributed notification service REBECA, which was partly implemented as part of the work presented in [Müh02] and is still under development. We give an comprehensive overview and detail some aspects of its realization. In the following Section 9.3 we clearly show how support for mobile clients can be added and how the relocation protocol detailed in Chapter 5 can be integrated into REBECA. This is done by adding the necessary support for relocating clients mainly in the event brokers at the borders of the system. We designed this in a way that mobility mostly is masked from the inner core of the distributed event routing network. Thereby, the existing infrastructure can be leveraged for the relocation process without hindering conventional message routing. However, an orthogonal extension is made in the core of REBECA for implementing the stateful relocation protocol (cf. Section 5.3) on top of the inherent stateless message routing. Finally, in Section 9.4 we give details on the integration of location-dependent subscriptions into the routing network. Here, special emphasis was laid on the implementation of a hybrid location model resembling the one introduced in Chapter 4, together with the algorithm detailed in Chapter 6.

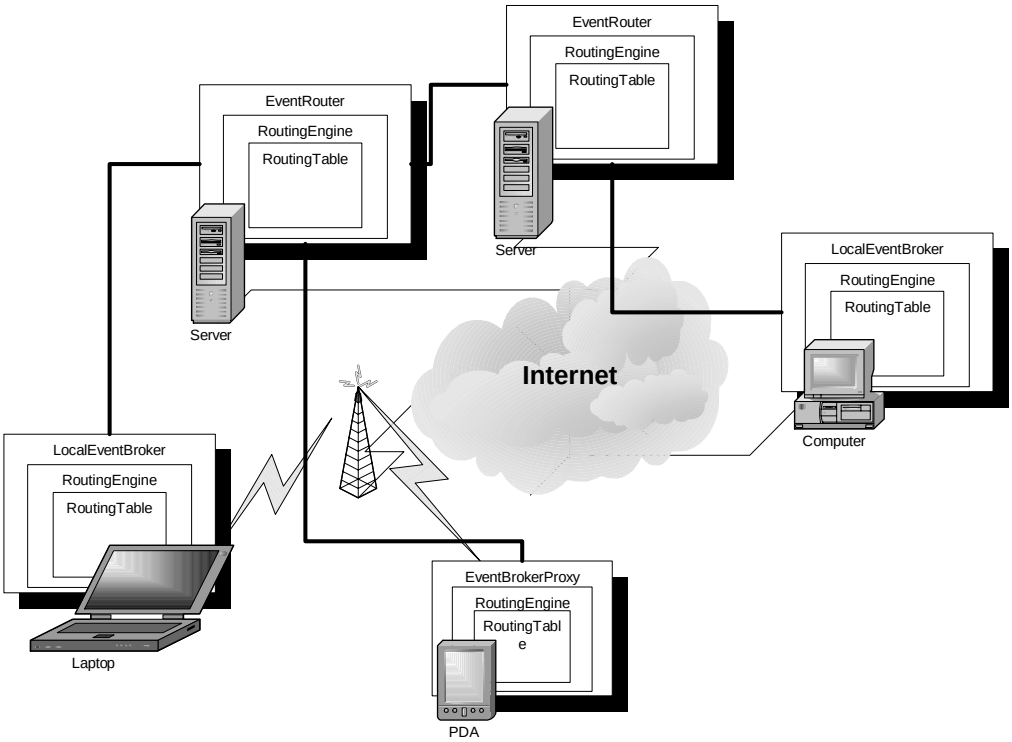


Figure 9.1: The REBECA routing network: spanning tree structure over physical networks

9.2 A REBECA Walkthrough

In this section we describe the implementation of REBECA which is the underlying code base for the extensions made to support mobile clients.

At the most abstract level the REBECA notification service consists of a set of communicating independent components, called event routers, running on different computing nodes in a network. Each of these event routers is connected to one or more other event routers, forming a *spanning tree* of event routers. This is detailed in Figure 9.1.

For clients to access the routing network of REBECA the interface `EventBroker` is specified (cf. Figure 9.2). It defines the most fundamental methods for accessing the system. Following the definitions and semantics given in Section 3.2 the main accessor methods are `publish()`, `subscribe()`, and `unsubscribe()`. Additionally, for publishers the methods `advertise()` and `unadvertise()` are introduced. They specify the ability to issue and to revoke advertisements for notification producers. The interface can be implemented in various ways.The two most important implementations in the basic REBECA framework are `LocalEventBroker` and `DefaultEventBroker`. We will add another one for mobile clients later in this chapter. The `DefaultEventBroker` is implementing the default behavior of an event broker. A `LocalEventBroker` is a specialized implementation usually

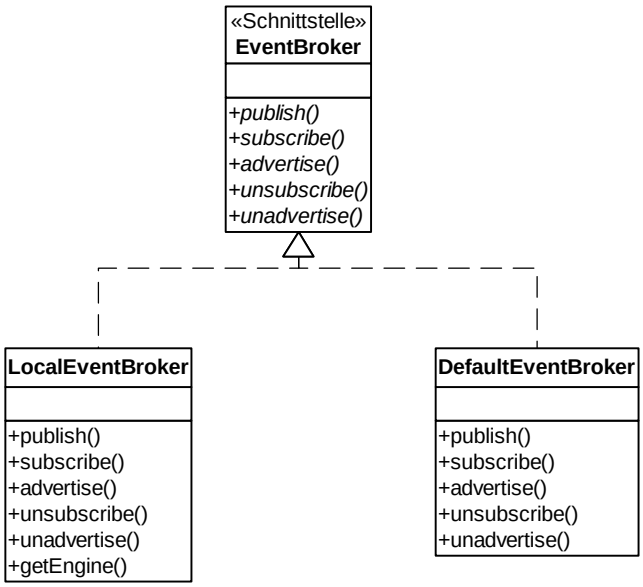


Figure 9.2: EventBroker interface and two important implementations

found on client machines. It is loaded onto the system as class library and provides the client with means to connect to the broker network. Internally, the local event broker acts as event dispatcher to the applications running on the client. Therefore, its behavior is different to an event broker running in the network.

Once a notification or subscription enters the routing network, it is forwarded towards appropriate addressees. Within the implementation, this is the task of event routers. Their basic behavior is to inspect a notification and compare it to subscriptions held in a table structure. Whenever a subscription matches a notification the event router forwards the notification in the direction indicated in the table entry. A similar behavior is implemented for subscription/advertisement pairs.

However, the implementing class `EventRouter` basically is a wrapper class managing the startup and initialization of a new instance of an event router on a particular computing node. The original design goal for REBECA is to provide efficient means for content-based routing, as introduced in Section 3.3.3. Therefore, the actual routing is delegated to specializations of a fundamental routing class `RoutingEngine` (cf. Figure 9.3). The specializations currently implemented for the generic content-based routing are the classes `SimpleRouting`, `IdentityRouting`, `CoveringRouting`, and `MergingRouting` that build up a hierarchy of inherenting classes. The semantics of using a delegate like `SimpleRouting` is that of providing a plug-in mechanism for new or specialized routing algorithms. The role of an instance of a `RoutingEngine` or its specializations is the processing and dispatching of incoming events, i.e., of instances of the general class `Event`, according to the implemented routing strategy. Thereby the REBECA model is easily extensible to new implementations of routing algorithms.

For the actual routing decision an implementation of a `RoutingEngine` refers to an instance of the type `RoutingTable`, which holds all subscriptions (realized by instances of the class `Filter`)

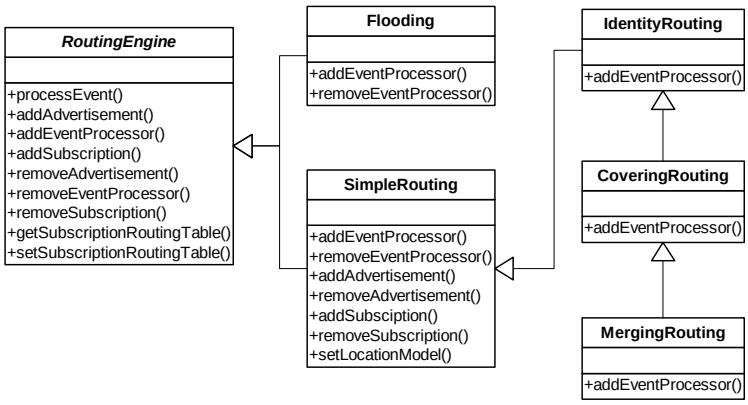


Figure 9.3: The class RoutingEngine and its specializations

and the associated connections to the next hops in the broker network. We have shown the class `RoutingTable` and its dependencies in Figure 9.4. `Filters` and `EventProcessors` are contained in a wrapper class `RoutingEntry`. To process a notification the method `processEvent()` in the class `RoutingEngine` is called. This method takes all steps necessary to forward or discard an notification according to the routing strategy and the filters used.

The method `match()` in the class `Filter` must be pointed out, as this method determines whether a notification *matches* this particular instance of a filter or not. `match()` returns a simple boolean value for this purpose. Thereby, the routing decision basically is delegated into the actual instance of a filter. This introduces an elegant means to provide various different decision functions for general and specialized application cases without limiting the generality of the routing network. We leverage this core design for implementing location-dependent filters later in this chapter.

Another interesting design decision is the interface `EventProcessor`, which is also shown in Figure 9.4. Once the filter has decided to match a particular instance of the class `Event`, the method `process` on the associated implementation of this interface is called. This causes the event to be processed further down the communication stack. The most important implementation to name are `EventTransport` and `EventSocketTransport`. `EventTransport` implements the core functionality and the most common specialization is the `EventSocketTransport` providing the needed functionality to send notifications over a socket connection.

In Figure 9.5 the class `Event` and its specializations are shown. `Event` is the base class for every notification. The complementary class to this event class is the base class `Filter` as shown in Figure 9.4. Any notification in the network is a specializations of the base class `Event`. This includes predefined "helper" classes, like `StringEvent` or `TimeEvent`. Three additional specializations are `ReplayEvent`, `ContainerEvent`, and `AdminEvent`. Especially `AdminEvent` and `ContainerEvent` play an important role as they provide means to convey administrative messages within the notification service. For example, the handling of subscriptions and advertisements is based on this mechanism: appropriate notifications are send through the network to indicate that a subscription or advertisement was added or revoked. The class `ContainerEvent` additionally allows to transport arbitrary content as content type `java.lang.Object`. This can be leveraged to

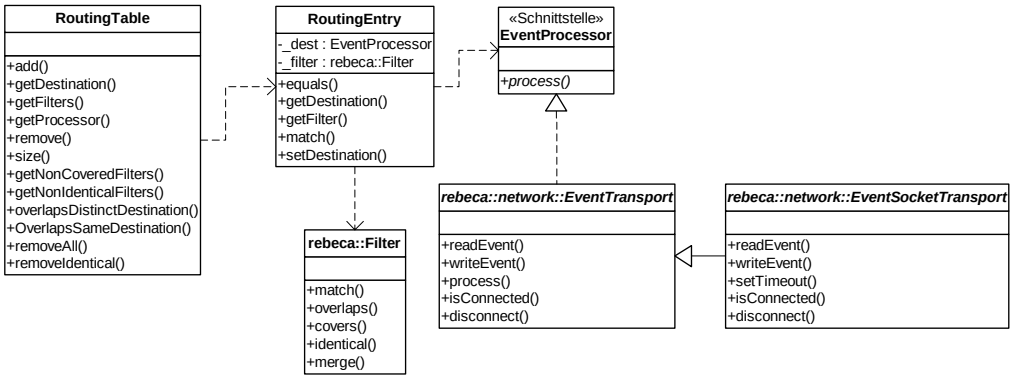


Figure 9.4: The class RoutingTable and its dependencies

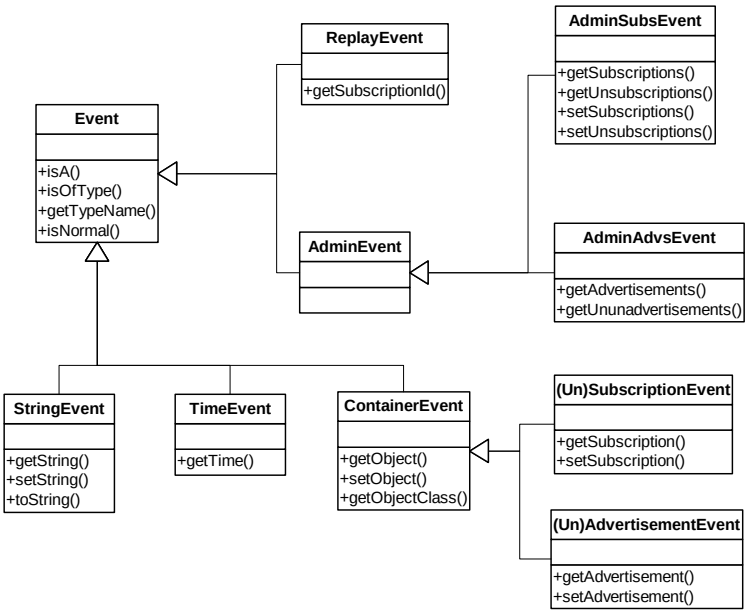


Figure 9.5: The class Event and its specializations

transport, e.g., a new subscription (as type `Subscription`) to other routers in the network.

9.3 Adding Support for Mobile Clients

Adding support for mobile clients as detailed in Chapter 5 is mainly a software engineering issue of adding functionality to REBECA without impairing the underlying functionality of the system. The algorithm we presented in the named chapter solely relies on main characteristics already present in the basic implementation of the REBECA notification service. The necessary changes made therefore leave the core functionality intact and are located either at the "borders" of the system or orthogonal to existing functionality.

The proposed algorithm explicitly uses the REBECA notification routing network for the relocation of clients from one access point to the system to another, i.e., the border brokers. Hence, at this point we had to decide which additions to the already implemented system are necessary to facilitate such relocations. As it turned out the mechanism to relocate clients by leveraging the routing network is beneficial with respect to the effort necessary to implement such functionality.

9.3.1 Meeting the Failure Semantics of REBECA

One important aspect that must be considered is that REBECA has no concept of frequent and regular disconnections and reconnections of clients. The underlying assumption for the original implementation of REBECA stems from classical distributed systems design. Consumers and producers are assumed to remain connected to the network permanently and in case of a disconnection all associated resources can be freed instantaneously because the client "died."

This semantics assumes a particular failure-mode: in case a client is not reachable anymore it will remain in this state forever. A client that connects to the notification service is assumed to be a new instance of a consumer or producer, respectively. As we have detailed before this assumption does not hold for mobile systems.

As a consequence, this results in the need to translate the semantics of mobile systems to the failure semantics used in the REBECA notification service. The task at hand is to make connections and disconnections of mobile clients opaque to the notification service and only disconnect clients from the broker when a permanent disconnection can be assumed.

One choice to implement such behavior is to simulate the proper behavior of a REBECA client on top of the semantics found in mobile systems. For this it is important to hide the fact of disconnection and reconnection from the broker network and delegate the proper handling to a special component. This is realized through an additional class `ConnectionManager` in the border brokers and a subclass of the `EventTransport` class: `DurableEventTransport`. Disconnection (in terms of REBECA) only occurs if (a) the client is relocated to a different location or (b) some timeout is reached such that the client is assumed to have died. In case of (a) the relocated client is unsubscribed from all its subscriptions and allocated resources can be garbage-collected safely. However, case (b) is problematic. Finding a reasonable threshold for garbage-collection is always a difficult choice to make. On the other hand, garbage collection has to be facilitated to free resources for other (mobile) clients.

For the implementation two objectives have to be met: (i) simulate the proper delivery of notifications to a client even if the client is currently not available and (ii) provide means to discern reconnecting, relocating and new clients at a given broker.

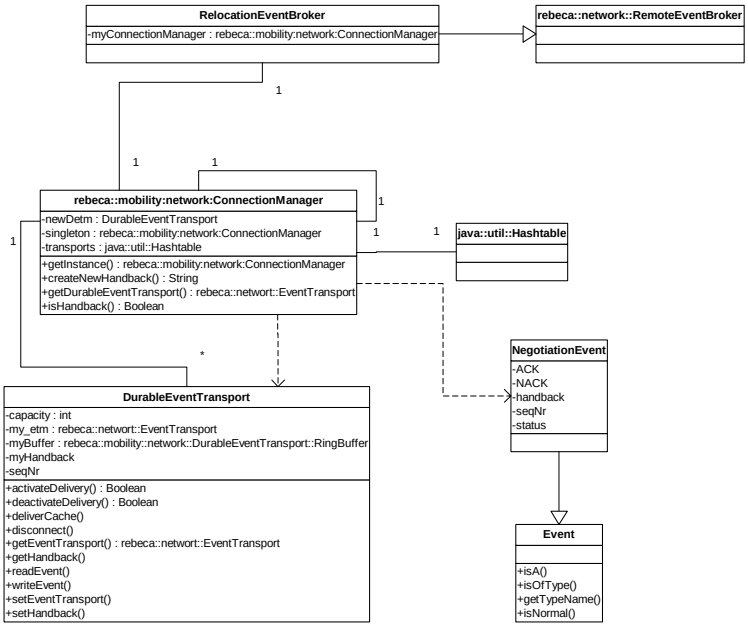


Figure 9.6: The class `RelocationEventBroker` and `ConnectionManager`

A new class `RelocationEventBroker`, as an extension of an `EventBroker`, is a direct subclass of the `RemoteEventBroker` (cf. Figure 9.6). It provides for the semantic wrapper of the concept described. The only addition to the fundamental behavior of a `RemoteEventBroker` is that the new event broker is aware of relocations. The `RelocationEventBroker` is responsible for the proper handling of mobile clients. Upon initialization it instantiates the singleton class `ConnectionManager`.

The `ConnectionManager` then is responsible for the proper management of client connections. Each broker instance has exactly one reference to a `ConnectionManager`. This single instance can be retrieved by calling the static method `ConnectionManager.getInstance()`, which returns the current instance of the class or creates one if this is the first call of the method. The main task of the connection manager is to discern the different flavors of client connections (new clients, reconnections, and relocations). It holds a list of all currently known clients in a hashtable data structure with unique identifiers as keys. Associated with each client is an instance of a `DurableEventTransport` (shown in Figure 9.6 and detailed in Figure 9.7). Once the connection manager has decided that a new client has connected, it instantiates a new `DurableEventTransport` and adds the client to its hashtable of new clients. However, deciding if a client is new, reconnecting, or relocating is done by means of a negotiation protocol (cf. Figure 9.8). A subclass of `Event`, `NegotiationEvent`, is leveraged to negotiate a connection. We chose this particular "transport encoding" for negotiation for three reasons: (a) we do not have to provide means for "out-of-band" communication, (b) notification handling is a built-in functionality of both sides of the negotiation protocol and therefore can be handled more easily, and (c) "legacy" clients which are not aware of mobility, by default, simply discard such a notification as they cannot handle them. This leaves the underlying functionality as a

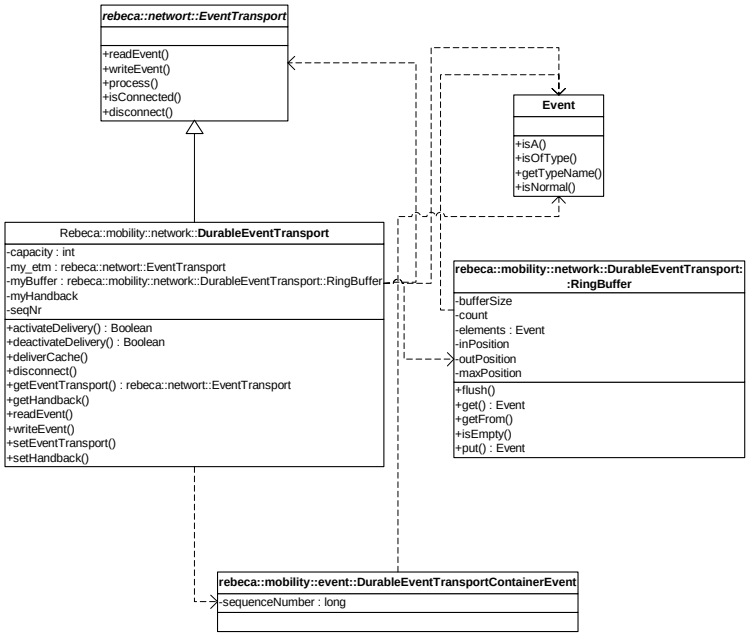


Figure 9.7: The class `DurableEventTransport`

"classical" notification service intact. In the following we give some details of the protocol.

Upon a connection request on a socket (stemming from the original implementation of REBECA) the connection manager instantiates a new `EventSocketTransport` for communication with the client. Then a new `NegotiationEvent` is created with a newly generated *handback*, i.e., a unique identifier as a token to identify the various steps of the process. We have roughly sketched the interaction between the objects involved in Figure 9.8 on page 153. At the core of the protocol the connection manager can discern clients by the handback used in the negotiation event. At the beginning of the negotiations the connection manager does not know whether a client is new, reconnecting, or relocating. Therefore, a new handback is generated and is sent to the client. The client inspects the handback and can decide its course of action: if the client is new the negotiation event is sent back with the member variable `ACK` set. This indicates that the client accepts the handback as unique id. When `NACK` is set[1], it sends back the event together with a (formerly assigned) handback.

Now the connection manager can discern all relevant cases:

- *New client.* In case of an `ACK` as response in the negotiation event, the client is considered to be new and a new `DurableEventTransport` is instantiated for communication. It is finally returned to the calling `RelocationEventBroker`. The client is added to the hashtable of known client, using the handback as key. As shown in Figure 9.7, the class `DurableEventTransport` basically wraps a "normal" event transport. Hence, the event transport used for sending the negotiation notification can be reused and is incorporated into its buffering variant (using the member variable `my_etm`).

[1] We are aware that using two different variables is highly redundant, but it is simply used for clarification.

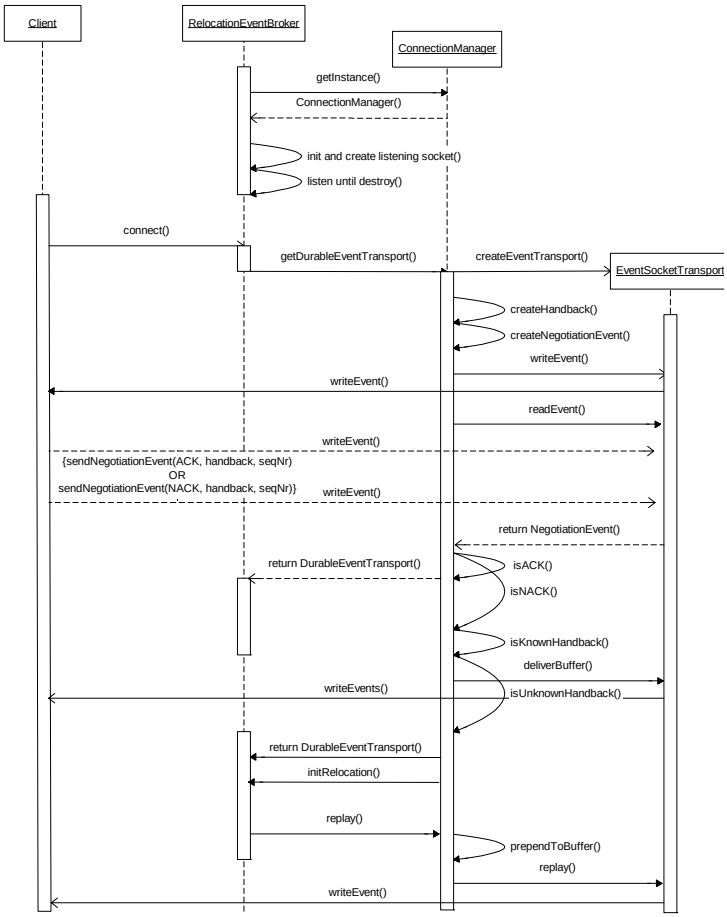


Figure 9.8: Interaction diagram of the negotiation protocol

- *Reconnecting client.* A reconnecting client can be recognized by the variable `NACK` set and a handback that can be found in the hashtable of known clients. Then the associated `DurableEventTransport` is retrieved, the encapsulated instance of an `EventTransport` is replaced by the one used for negotiation and together they are finally returned to the `RelocationEventBroker`. After activation the buffered notifications contained in the buffering data structure of the `DurableEventTransport` are delivered to the client.

- *Relocating client.* A relocating client also can be recognized by the `NACK` variable set. Additionally, the handback provided is not in the hashtable of known clients. In this case the relocation of the client has to be initiated. The fundamental behavior is the same as in case of a reconnecting client. A new `DurableEventTransport` is used and added to the list of known clients. This reference is finally returned to the caller. After returning the reference the client is “active” as far as the event broker is concerned. However, prior to the termination of

the relocation process, delivery of notifications in the `DurableEventTransport` is set to hold (cf. the description of the underlying relocation algorithm in Chapter 5). This is done to assure FIFO ordering. After receiving the subscriptions of this client, `RelocationAdminEvents`, as specializations of `AdminEvent` and `ContainerEvent`, are created and handled to the event broker for further processing. The `RelocationEvent` constitutes the basic semantics of the relocation process. They contain a subscription of a client, the sequence number of the last heard notification and the handback provided to identify the client at other brokers.

The endpoint of the negotiation in each case described above is returning a reference to a `DurableEventTransport` to the calling `RelocationEventBroker`. Moreover, in case of a relocation, the connection manager submits `RelocationEvents` for further processing.

The main addition made to the `RelocationEventBroker` and (as described below) the event routers in the network is the proper handling of relocation admin events. Briefly, each broker or router had to be extended for proper handling of such events and must take certain actions that we specified in the relocation algorithm in Section 5.1. Summarized, each broker has to add additional fields to its routing table holding information about the identity of a client for a subscription and the status of an ongoing relocation. Therefore, message routing is not completely stateless.

9.3.2 Embedding the Relocation Process into Stateless Message Routing

Within the core of the REBECA network some adaptations were made. The central problem we face is that the relocation protocol requires some state information to be hold in the event routers. This contradicts the paradigm of a distributed notification service where message routing is intrinsicly stateless. Our algorithm requires that a subscription of a client can have different states throughout the progress of the relocation process. Moreover, we use a handback to map subscriptions to clients. This is also additional information to be maintained in the routers. However, as long as the changes required for hosting mobile clients are completely orthogonal to the underlying paradigm of message routing in the classical settings, we consider this to do no harm. Two migration paths from a normal REBECA network to a mobility-aware REBECA network are feasible: first, each router in the network is replaced by a new router, having the necessary facilities for relocation handling, or second, a completely new overlay network is built parallel to the existing one. For simplicity, we assume the first approach.

The basis for our extensions is the class `EventRouter` as described in Section 9.2. This class provides flexible means to "plug-in" new implementations of the class `RoutingEngine`, or more precisely its specializations. We chose to leverage this concept for the two main changes to the current REBECA system we describe here. First, we adapted the routing engine for simple routing to proper handling of instances of `RelocationAdminEvent`, the administrative event we introduced for handling the relocation process. Second, we extended the classes `RoutingTable` and `RoutingEntry` for holding state information about mobile clients. The roles of those classes in the REBECA notification service are detailed in Section 9.2 and are shown in Figure 9.4.

As a first step, the existence of a relocation process must be conveyed. This can be done by introducing a new administrative notification `RelocationAdminEvent` extending the already introduced `AdminEvent` class. However, we had to change the underlying `RoutingEngine` for the *simple routing* strategy accordingly. Therefore, we changed the existing implementation and refactored the current implementation such that further extensions can be integrated more easily. We chose to change the whole handling of `AdminEvents` to a *Listener Pattern* approach for the integration of the new

type of event into the event router. Moreover, for further extensions we prepared the implementation for employment of *factories* whenever an (currently) unknown event type is encountered. But, a full implementation is left for future work. Goal of the redesign of the `EventRouter` class is that for a particular type of `AdminEvent` handlers can be added or removed on-the-fly and appropriate actions can be associated to these listeners. This resembles the *ActionListener* concept as known from the standard Java2 GUI APIs.

The next changes concern the routing engine and even more the included routing table of currently active subscriptions. Here, we made changes for holding state about the current status of the relocation process. For normal operation it is enough to hold information about the subscription and the associated event transport in the routing table. For the extensions we made, additional information is necessary. Whenever a event router receives a administrative event about a relocating client it has to check its routing table for the handback provided in the notification. As detailed in Section 5.4, the main goal within the relocation process is to identify the junction where old and new delivery path of the client meet. Therefore, two cases can occur: (i) the routing does not hold information about the client, hence the current event router is on the new delivery path, or (ii) the routing table holds such information. In case of (i) the event router adds the (new) subscription to its table together with the handback. This is new to the implementation. In case of (ii) the first router which holds information about the client is the router "sitting" on the junction. It marks the entry for the client with a flag indicating that the client is relocating and creates a new relocation event with a special marker set, indicating that this new event is sent by the router at the junction. The semantics is that of the "fetch" message described in Section 5.4. Finally, it adds its own router id to the message. This is later used for identifying the last router with a junction (cf. to Section 5.3.3 for details).

The new message is then handled to the event transport for this client retrieved from the routing table. As it belongs to the "old" delivery path, it points into the direction of the old border broker. After this, the event transport is removed and replaced by the event transport the relocation message was received from and message delivery resumed. Thereby, any newly received notification matching the subscription for this clients is already delivered to the new location and buffered there for delayed delivery after the relocation process has terminated. The further progress is straightforward and detailed in Section 5.4. Each router on the "old" delivery path inspects its routing table, retrieves the routing entry for the relocation client, retrieves and replaces the event transport in the entry and sends the relocation message into the old direction. Additionally, in case of another junction identified, the router id in the message is replaced by the own id.

Upon receiving a relocation message, the border broker at the old location of the client also retrieves the event transport from its routing table (please note that this particular event transport is an instance of the `DurableEventTransport` class) and accesses its buffer. The messages are then replayed using a specialization of the class `ReplayEvent` stemming from the original implementation of REBECA. The messages are then replayed in the direction of the junction. However, for safety and, more importantly, for garbage collection, the received fetch message is sent again "upstream." This message then is used as a closing event for this relocation process. Each broker on the path inspects the message and removes all resources allocated for this client and subscription. This is done until the first junction in case of a multiple producer scenario is reached. The broker on the first junction sets the "relocation finished" marker in the message and relays it further upstream. Those brokers receiving this message now can determine that the process terminated successfully and remove the relocation flag from their routing tables. The message is discarded when the new border broker is reached.

The new border broker then prepends the replayed notifications to the ones received in-between,

adds them to the associated buffer, and sets message delivery in the event transport to active. By design the `DurableEventTransport` then delivers the notifications to the client.

9.4 Location-dependent Notification Delivery

Here, we give some details on the implementation of location-dependent notification delivery as it was implemented in [Mac04]. It serves as proof-of-concept for the algorithm presented in Chapter 6.

The implementation has the goal to show that it is feasible to integrate location information into the core of the REBECA routing infrastructure. Therefore, we chose to use a location framework as presented in Chapter 4 and integrate it into REBECA. As detailed before, the location model is comprised of two parts: the hierarchical symbolic location model as it is used with mobile clients and the geometric location model for efficient notification/subscription matching for the use in the routing network. The requirements for the implementation of this model can be summarized as:

- *Hybrid model.* The model we specified can express geometric as well as symbolic location specifications. This has to be considered in the implementation. Both should be possible at the same time.
- *Semantic information.* Clients of the network possibly want to express location specifications containing semantic information meaningful to an certain application domain. This is part of the symbolic location model where strings of text denote a location, e.g., *room* or *floor*.
- *Efficient subscription/notification matching.* Symbolic locations describe a certain area according to an underlying semantic model. However, the most flexible model for the use within the routing network is a geometric model. Therefore, to include the best-of-both-worlds, mappings from one domain to the other have to be integrated. The existence of such mappings is shown in Chapter 4. In the implementation such mappings have to be implemented at the borders of the network.
- *Implementation of the `myLoc` marker.* As part of the location model we introduced the special marker `myLoc` as a generic reference to the current location of a client. The update and evaluation of this marker is delegated to the broker network. Handling of subscriptions holding the `myLoc` marker therefore is a requirement for the implementation.

9.4.1 Location model

The hybrid location model is comprised of two parts: the symbolic and the geometric model. For the geometrical model it was chosen to specify the underlying characteristics on the basis of a mathematical model for geometric primitives. The fundamental aspects of those are prototyped in a set of abstract foundation classes. They realize the characteristics inherent in each specialization of the underlying model.

This approach provides an elegant means for implementing mappings from the symbolic model to the geometric model. They can access any actual geometric model by means of using the methods specified in the abstract base classes. Hence, a separation of concerns is provided, as the symbolic model does not have to be aware of the particular characteristics of the specializations of the abstract classes. For demonstration purpose the model we chose for specialization is a simple model of two-dimensional primitives.

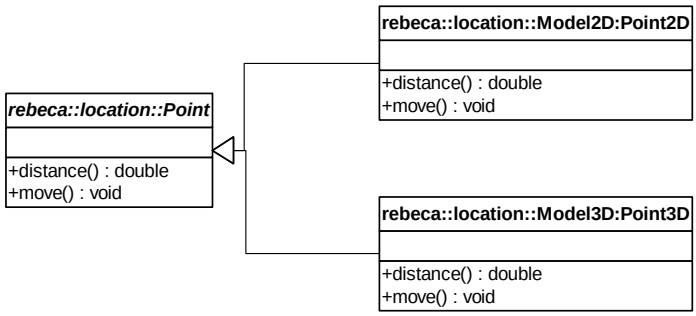


Figure 9.9: The abstract class `Point` and two possible specializations

One characteristics for the description of a geometric location is the existence of a reference point. This is concretized in the abstract class `Point` as shown in Figure 9.9. The interface to the class `Point` has two methods: `distance(Point)` and `move(Vector)`. The method `move()` "sets" the point to the coordinates specified in the argument. Please note that by using a dynamic data structure for this specification, arbitrary arguments can be used for coordinate specification. Thereby, specializations of the model can use their own descriptions or number of coordinates contained in the vector. The other method provided is `distance(Point)`, returning the distance to another point. Around this minimal abstraction higher level of shapes can be build.

As a foundation for building up more complex geometric figures the base class `GeometricPrimitive` is introduced. We showed its UML description and its most important relationships to other classes in Figure 9.10. The class `GeometricPrimitive` is meant as the most basic building block for geometric shapes. For ease of use this class already holds methods for testing containment of and overlapping with other geometric primitives. The method `contains()` is overloaded to test if a point is contained within the given geometric primitive. Additionally, methods are provided for the management of primitive geometric objects. These are `move()`, `moveTo()`, `getCenter()`, and `getBoundingBox()`. They have the obvious semantics and details are left out here. The only method more remarkable is `getBoundingBox()`. When called an object of the (abstract) type `BoundingBox` is returned. A bounding box is used to speed up notification/subscription matching within the routing infrastructure. A bounding box is a simple geometric shape approximating the contained complex shape. Usually boxes or circles are used as bounding boxes because inclusion or overlapping can be tested easily. However, using a bounding box obviously introduces imperfect matching as regions of shapes might overlap when using a bounding box but were not overlapping if perfect matching is used.

We chose to include this approach as we consider performance of notification matching in the infrastructure more valuable than network traffic generated by imperfect matches. In the worst case, finally at the last broker on the path towards a client perfect matching is applied and therefore false matches are eliminated. This is reasonable because clients often are connected to the network via low-bandwidth wireless connections. Here, it makes sense to reduce network traffic as much as possible as well as relieve the client from discarding false matches by itself.

The last abstract foundation class we introduce here is the class `GeometricShape`. It is the core abstraction for defining complex geometric shapes. This class has to be specialized for describing

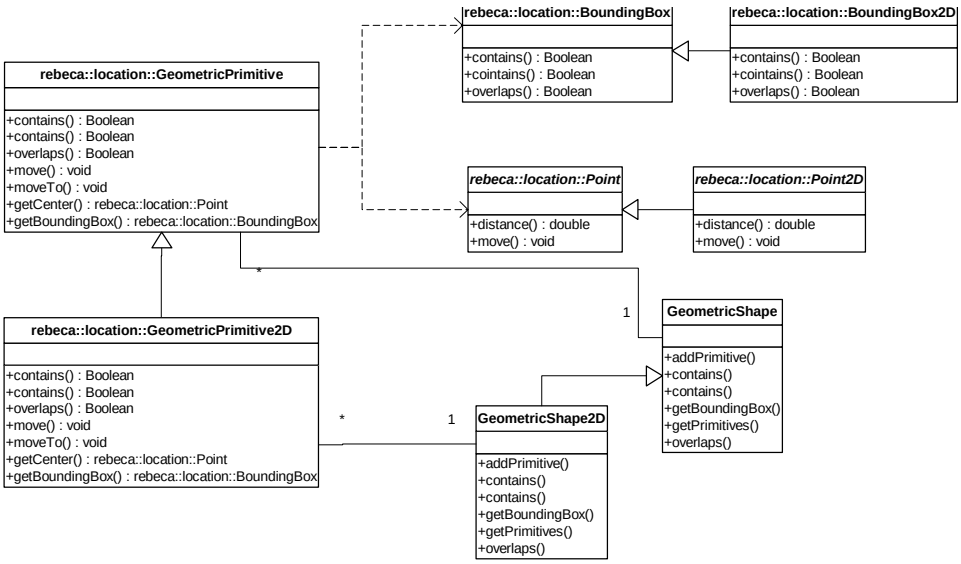


Figure 9.10: The abstract classes `GeometricPrimitive` and `GeometricShape`

real-world objects in a certain geometric model. The semantics of this class is to provide a uniform wrapper for accessing instances of `GeometricPrimitives`. It semantically wraps a number of geometric primitives, which together represent a certain object in the real world. Therefore, the class holds methods for (a) evaluation of overlapping and containment and (b) methods to add and retrieve the wrapped primitives. The methods for (a) are `contains()`, again as overloaded variant with either `Point` or `GeometricShape` as parameter, and `overlaps(GeometricShape)`. Methods for (b) are `addPrimitive(GeometricPrimitive)` and `getPrimitives()`, with the obvious meanings. Each instance of `GeometricShape` holds a list of (lower level) geometric primitives and together they form a geometric shape which corresponds to an object in the real world.

For testing overlapping and containment an external object calls the appropriate method together with the shape to be tested against the shape called. Instances of `GeometricPrimitives` included in the shapes are tested pairwise by calling the delegate methods `contains()` or `overlaps()`, respectively, in the `GeometricPrimitive`. For overlapping it is sufficient if one of the tests is true, i.e., at least one geometric primitive overlaps with an area described in the second primitive. For containment, each geometric primitive of the one shape must be contained in one or more primitives of the other shape.

As all the classes described above are abstract, for using them at least one specialization must be implemented. For demonstration purposes we have depicted one possible specialization in Figure 9.10: an implementation for the handling of two-dimensional shapes. This model is sufficient for, e.g., the handling of movement of people on the basis of floor plans as introduced earlier in this thesis.

In order to make the classes described above more easily to handle, the class `GeometryManager` was introduced. One of its main tasks is to instantiate geometric objects when needed. The class

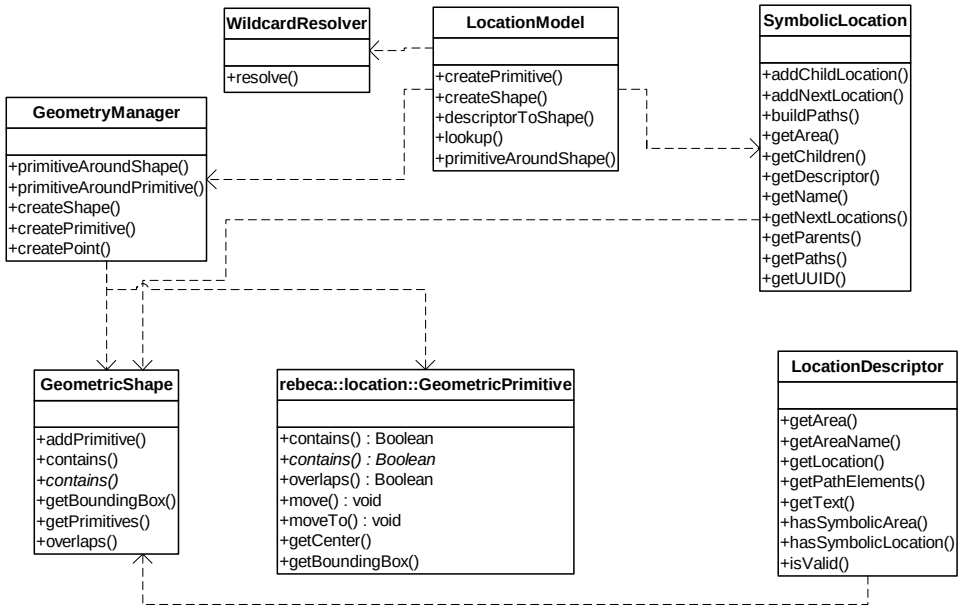
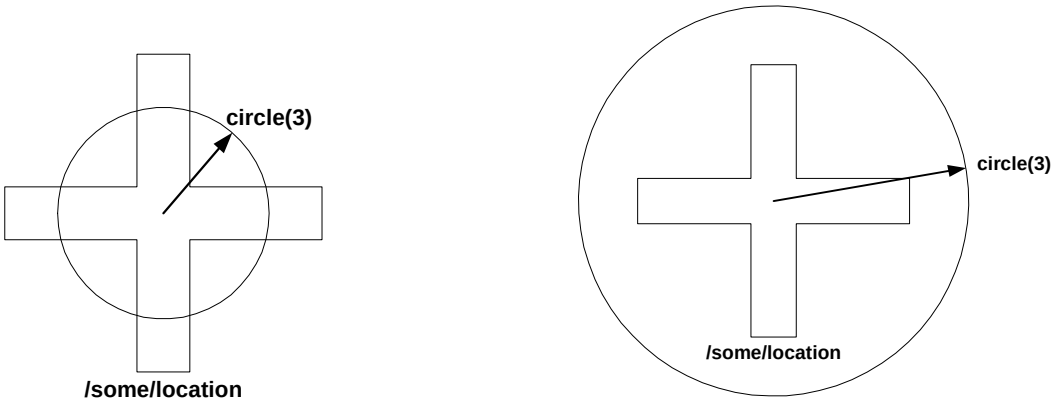


Figure 9.11: The class `GeometryManager`, `SymbolicLocation`, and `LocationModel` and their relation to the geometric model

`GeometryManager` also is responsible for creating geometric shapes out of a textual description, i.e., is the main access point for translating symbolic locations into the geometric representations used within the infrastructure.

The methods `createPoint(String)` and `createPrimitive(String)` create a point or a primitive geometric shape out of a textual description as specified in the symbolic location model. The method `createShape(List)` creates an object of type `GeometricShape` out of the List of geometric primitives contained in the `List` object in the argument of the method. More remarkable are the two methods `primitiveAroundPrimitive(String, GeometricPrimitive)` and `primitiveAroundShape(String, GeometricShape)` of the class `GeometryManager`. They return objects which encapsule the primitive or shape given as second parameter according to the textual description given as first argument. This is needed for realizing textual descriptions like "circle(3)@/some/location". "/some/location" is translated into the corresponding geometric shape and then a new circle is generated that encloses "/some/location." However, at this point, we are aware that location descriptions like the one above are semantically problematic. With respect to the specification "circle(3)" it is apriori unclear to what part of "/some/location" it relates to. Possible choices are: (i) the center of "/some/location" (such that the resulting circle might be contained in the location specified by "/some/location"); or (ii) the point with maximum distance to the center of the location and then plus the distance specified in "circle(3)," resulting in a circle larger than "/some/location" by design. We have illustrated this in Figure 9.12 on page 160.

However, we deliberately chose to not preclude this ambiguity but to delegate the specification of the actual semantics to the specializations of the location models according to their special needs.



(a) Relative to center

(b) Relative to border

Figure 9.12: Possible semantics of location specifications

In the simple two-dimensional model we chose as a proof-of-concept realization the semantics is defined as shown in Figure 9.12(b), i.e., relative to the border of the location specified. Future work can extend the model in a way that the behavior can be parameterized to accommodate the actual need at runtime of the system.

Finally, in Figure 9.11 on page 159 we have shown the foundation classes for handling symbolic location specifications. The most important classes are `SymbolicLocation` and `LocationModel`. The class `SymbolicLocation` encapsulates the behavior of a symbolic location as specified in Section 4.3.4 for the location domain model. There, a symbolic location model is specified which can be organized as tree or lattice structure by using the inclusion relationship. Consequently, the classes representing the symbolic model have methods reflecting the creation and use of this hierarchical structures. The methods `getParents()` and `getChildren()` are used for the traversal of the inclusion graph. For a given `SymbolicLocation`, they return the list of parent (or children, respectively) `SymbolicLoactions`. This is used for mapping geometric locations to their symbolic counterparts and for resolving path descriptions.

A hook to the geometric model is provided by the method `getArea()`, which returns the geometric representation of the specified symbolic location as an object of the type `GeometricShape`. `getName()` and `getDescriptor()` return the name of the location or the symbolic description, respectively. The former is used in conjunction with symbolic specifications in subscriptions, the latter for correctly resolving specifications like “room@/some/location.”

The method `getNextLocations()` is the concretization of the *movement graph* as introduced in Section 4.4.4. It returns a list of `SymbolicLocations` a mobile object probably can reach in the near future. We have not laid great emphasize on adaptive behavior and heuristics for the prediction of probable future locations of a client. The implementation of this method is rather static and straightforward. For future work, obviously, this is an issue worth spending some time on.

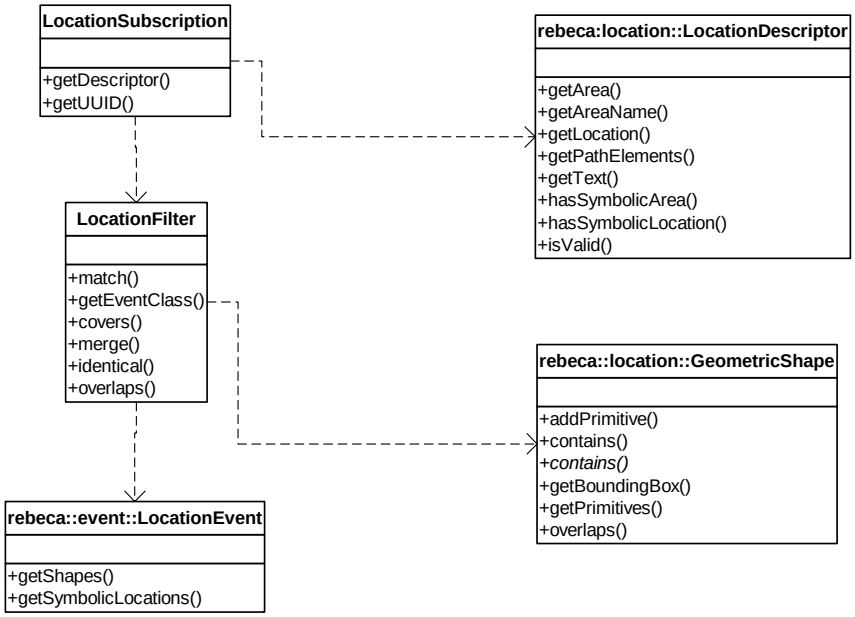


Figure 9.13: Main classes for location-dependent subscriptions

The last class we want to introduce here is `LocationModel`. It integrates the classes described above into a complete model and serves as central entry point to the system. Therefore, the realization is done using the *Singleton* design pattern, i.e., at any given time only one instance of the class is active and can be accessed by the static method `getModel()`, which returns a reference to the active object. This way, access to the model can be realized from arbitrary parts of the system. The method `lookup(String)` returns a list of symbolic locations matching the location specification provided. As we allow wildcards, more than a single location can match the specification.

For the mapping of symbolic descriptions to geometric objects the method `descriptorToShape(LocationDescriptor)` is provided. It maps the given `LocationDescriptor` to its corresponding geometric objects. A `LocationDescriptor` encapsulates an expression of the location description language and provides for easy access. Contrary to the more simple `lookup()` method of `LocationModel`, complex path descriptions like "hall@/*/piloty/*/" can be mapped to the geometric model by a single method call.

Additional methods are provided in `LocationModel` for handling geometric primitives but they are simply delegated to a corresponding `GeometryManager`.

9.4.2 Location-dependent Routing

For location-dependent message routing it is necessary to introduce a set of new filter classes into the filter model of REBECA. These are shown in Figure 9.13. Naturally, there exist the usually complementary set of subscription, filter, and event for location-dependent routing. In this concrete case these are the classes: `LocationSubscription`, `LocationFilter`, and `LocationEvent`. In Fig-

ure 9.13, we have also shown the dependencies of this classes in order to illustrate the relationships between the filter model and the location model.

A location-dependent subscription enters the system by means of an instance of type `LocationSubscription`. The location specified can be retrieved by calling the method `getDiscriptor()`. This method holds the hook to the location model in use, as it returns an object of the type `LocationDescriptor` which directly can be accessed to map the specified symbolic location to a corresponding geometric representation by calling the `getArea()` method provided.

This information is then used to create a `LocationFilter` adhering to the filter semantics used within REBECA. This class provides minimally the usual `match()` method. Additionally, it can provide methods for more advanced routing strategies, indicated by the methods `covers()`, `merge()`, and `identical()`, corresponding to the appropriate routing strategies with the same names. However, one additional method is necessary to facilitate for location-dependent routing, namely the method `overlaps()` which tests for overlapping between a filter and a notification. Here the design of the location model pays well, as this call is directly delegated to the appropriate object of the type `GeometricShape` which does the actual testing (cf. Figure 9.13). For every location model used the interface for testing overlapping remains the same.

Location-dependent routing remains the same as in the standard case. An incoming notification N is tested against the subscriptions in the routing table of a message router. If the subscription matches the notification, it is forwarded as indicated in the routing table via the associated event transport. As described above, testing for matching is done simply by calling the method `overlaps(GeometricShape)`.

An additional extension was made for mobile devices with limited resources available. The hybrid location model makes it necessary to map symbolic locations as used within a mobile application to a geometric representation. We consider this translation step as being too complex to require a mobile device to do it itself. Therefore we devised an intermediary, providing an event broker interface to the application running on a device, on the one hand, and delegating the actual translation to the first hop in the infrastructure, on the other hand. The class `ClientRouterConnection` implements such behavior. To the client it acts as a local event broker, but to implementations of `EventRouter` it acts the same as the standard implementation of `RemoteEventBroker`. Therefore, a client can use symbolic descriptions within its subscriptions without the need to translate the subscription into a geometric representation. This is then done in the border broker the client is connected to. There a location-dependent subscription is handled as follows: (i) the subscription already carries a geometric location, then the subscription is simply forwarded according to matching advertisements. (ii) the subscription only carries a symbolic description. Then the symbolic description is resolved by calling `LocationModel.descriptorToShape(LocationDescriptor)` and the geometric representation is added to the subscription. The symbolic subscription remains part of the subscription and is piggyback for potential further use. After the translation step the subscription is handled as in (i).

For the use together with the class `ClientRouterConnection` it was necessary to introduce the second step in the border brokers of the system. Otherwise it would be the responsibility of the local event brokers on the devices. By handling both in the border brokers, either solution is possible.

9.4.3 Routing with Mobile Objects

In Chapter 4 we defined the specification language for subscriptions in a way that it is possible for mobile clients to express their interest in information related to the current position. The inherit

semantics is that the subscription virtually "moves" when the client moves. This way, the client always only receives information matching the location-dependent subscription and therefore is valid for the current position.

We bound this special semantics to the special marker `myLoc` as part of the specification language for subscriptions. As the name suggests, it then acts as a placeholder for the current position of a client. The most basic use is in subscriptions of the form "area@MyLoc." However, as discussed earlier the problem arises how this marker is actually resolved to the position of a client. We specified the semantics in a way that the responsibility for updating the current position is delegated to the notification service and part of its subscription processing. On the other hand, we consider location tracking and administration as an orthogonal concern to the work presented here and should not be part of a notification service. Therefore, as shown in Figure 4.9 on page 73, we assume an external source for such information.

However, an external location service has to be integrated into the notification service to some degree as it has to gain knowledge about new subscription with the `myLoc` marker, on the one hand, and it has to issue new location update events in case a new location for a client is detected. Without going into details, this integration easily can be achieved by using the well-known *observer design pattern* implemented by an appropriate observer for location changes for known clients. It then acts as event broker to the notification service (and therefore receives new location-dependent subscriptions) and registers for location changes of known clients with the external service. Every time a client's location change is observed, an appropriate `MyLocUpdateEvent` is generated and dispatched into the notification service. We have shown the corresponding class and its dependencies in Figure 9.14. In the following we assume facilities for observing location changes of mobile clients to be in place. Please note that we also have to assume that clients can be identified throughout a "session," i.e., we have to assume that the location service and the notification service have to have a common understanding about a client. The handbacks used in the previous section in general are not sufficient for this purpose. Therefore, for the sake of simplicity we assume Universally Unique Identifiers (UUID) for this purpose. In case of an explicit log-on to the network these can be generated automatically, otherwise the identifier must be provided by the client and accessible by the location service. This is the case, for example, when badges or RFID tags are used.

In the following we sketch the behavior of a broker receiving a location-dependent subscription from a client. The initial processing of the subscription of type `MyLocSubscription` is straightforward. It is comprised of the two main parts: (i) the identity of the client which can be retrieved by calling the `getUUID()` method on the subscription and (ii) the location-dependent subscription, accessible by calling `getDescriptor()`.

Obviously, if this subscription is the first subscription of this client at this broker, the UUID is unknown initially and therefore the current position cannot be resolved by the broker. Then, the subscription is stored in the broker for later activation. Additionally, the broker subscribes to notifications of the appropriate type `MyLocUpdateEvent` for this client. Thereby, eventually it will receive a notification with the last heard position of the client.

Please recall that the location service has an observer attached that acts as if it is an event broker and therefore eventually will receive this subscription for `MyLocUpdateEvents`. At the same time a delivery path between border broker and location service observer was built up. According to the UUID enclosed in the subscription the location service is queried and the current position retrieved. The location service observer then generates a new `MyLocUpdateEvent` and publishes it into the notification service. Each router on the path forwards the event into the direction of the border broker.

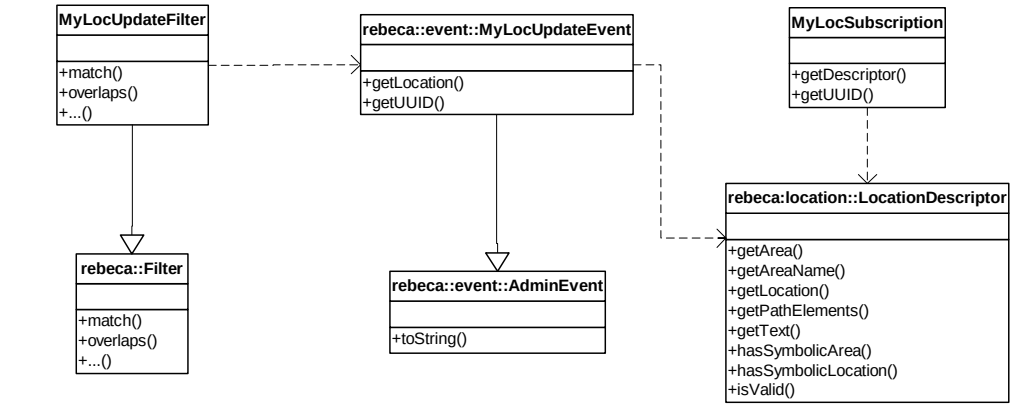


Figure 9.14: The `MyLocUpdateEvent` and its filter

Please note that without explicitly sending an initial `MyLocUpdateEvent` the client might never receive any notifications. The generation of such events is normally tied to a position change of the client in the real world. As long as a client does not move, no notifications are generated. To avoid such "starvation" an initial notification for the current position must be generated.

The behavior of a broker receiving a `MyLocUpdateEvent` is as follows:

- If the UUID in the `MyLocUpdateEvent` is not known, then process the message as any other notification, i.e., it is forwarded into the directions indicated by the routing table.
- Otherwise:
  - Call `LocationModel.lookup(locationString)` to resolve the location contained in the message.
  - If the old location is the same as the new one, do nothing.
  - Otherwise:
    - Retrieve and then store current *and* possible next locations of the client in a List L.
    - For each location loc in L: subscribe(loc), based on the original subscription of the client.
    - For each location loc' in the active set of myLoc subscriptions and that is not in L: unsubscribe(loc').
    - If the client has subscribed to *myLoc* updates: propagate the message to the client.

One aspect of the algorithm described above is that the broker not only subscribes to the current location of a client but also to locations which are probable next locations of a client. This realizes the concept of *uncertainty* as described in Chapter 6. The determination of the actual degree of uncertainty is delegated to the location model and independent from both, the client and the broker. Thereby, certain local peculiarities can be handled more easily and adaptive behavior can be implemented there. However, this partly is left for future work and the current implementation uses

a fixed behavior. We extended the class `RoutingTable` to optimize the behavior of routing with "deactivated" subscriptions. A subscription is deactivated if it belongs to the active set of *myLoc*-subscriptions of a client, but the client is not known to have moved there already. Therefore, the behavior of the extended routing table is changed accordingly. It can ignore currently deactivated subscriptions which avoids significant overhead for evaluating (and later discarding) notifications for such locations. The main advantage is that the broker does not have to test notifications before sending them to the client by calling the `GeometricShape.overlaps(GeometricShape)` method for deactivated entries in the routing table.

It is to note that again this extension is mainly part of the border brokers at the boundaries of the system. Within the broker network only geometric representations are used. The inner routers are not aware of the existence of a *myLoc* marker. Any translation necessary always is done in the border brokers at the system's boundaries. On the other hand, we chose to *piggyback* the original location descriptor to any notification and subscription mapped from the symbolic to the geometric model. Thereby, no information loss happens and a client can access any location information. A client which is interested in its own location and location updates can subscribe to its own location update notifications and a `MyLocNotificationEvent` is sent every time a location change is observed, holding the symbolic and geometric position information.

Finally, we want to sketch the behavior of a border broker in case of an unsubscribe to the last subscription:

- Unsubscribe to all subscriptions related to the current position of a client.
- Update the routing table and remove all deactivated entries for future positions.
- Unsubscribe to *myLoc* update notifications for this client.
- Remove any other management information related to this client.

9.5 Miscellaneous

The ideas presented in Chapter 7.1 and in particular the algorithm introduced in Section 7.2 currently are under development. Details will be available soon in [Gue04] and are omitted here.

A working prototype from the health care domain was developed using parts of the approach presented in Chapter 8. Details are available in [Rei02]. However, there the goal was to design an personal health care monitor application using the *ContextToolkit* API [SDA99]. Although vaguely related in some aspects to work presented in this thesis, the version of the API used for the prototype implementation was strongly based on synchronous and direct client/server communication. Therefore, details about the actual implementation do not add to this thesis and consequently are omitted. Nevertheless, the work in [Rei02] strongly showed the usefulness of the process of context abstraction and aggregation and was used extensively.

9.6 Summary

In this chapter we explained important details of the design and implementations done related to this thesis. As the basis of all extensions, we leveraged an existing implementation of a distributed

notification service, namely REBECA. Therefore, as a starting point, a comprehensible walkthrough in Section 9.2 gives an overview of its features.

Then we focused our discussion on two main aspects: first, the extensions made to facilitate generic support for mobile clients, which have to conduct their operations in an unstable and changing environment. And second, the changes necessary for facilitating location-dependent subscriptions.

We tackled these two aspects in separate sections. In Section 9.3, we showed how the REBECA notification service was extended in such a way that mobile clients can be used in conjunction with the core REBECA system. We grouped our discussion around the two main challenges hindering such integration. The first is that mobile clients regularly disconnect and reconnect to the network. This behavior constitutes a semantics that is contradicting the semantics found for "traditional" message routing in distributed systems. We clearly showed that by using intermediaries between the mobile system and the routing infrastructure this issue can be resolved and both semantics remain intact. As a side effect, the underlying functionality for conventional message routing is not impaired. The second issue tackled is that for relocation we have to superimpose a stateful process onto the inherent stateless message routing paradigm. Again, we gave an implementation that reached this goal with minimal changes made to the original design. The changes were devised in a way that they are completely orthogonal to the existing functionality.

We then showed in Section 9.4 how location information as a valuable source for context-dependent applications can be integrated into the very same notification service. The challenge here was to find a tradeoff between the usability of location-specification for clients and the requirements of efficiency as found in event systems. We therefore put to use our reference model for location specification as introduced in Chapter 4 and applied it to the problem at hand. To do so, we had to integrate it into the broker infrastructure, again as an orthogonal concern to the normal operation of the system. This could be achieved by leveraging the extensible model of "plug-ins" as specified by REBECA, on the one hand, and by restricting the necessary changes to the core functionality to happen at the boundaries of the system, on the other hand. Here, we refer to the mappings from symbolic descriptions to geometric representations and the special placeholder *myLoc*. This special marker relieves the clients from being aware of their movements and the otherwise necessary "manual" updates for their subscriptions. As required in Chapter 2, the infrastructure takes care of this.

To summarize, this chapter gives a clear picture of many aspects of the implementations done in the context of this thesis. Sufficient proof-of-concept systems were built, not only to show that the design choices made earlier in this thesis can be implemented, but also to show that the algorithms are specifically designed with two main aspects in mind: first, the underlying notification service's semantics has to remain intact and second, it also can be leveraged for the extensions necessary for mobile clients and their needs.

10 Conclusion

Great things are done when man and mountains meet.

William Blake, artist (1757-1827)

Many researchers and analysts expect the mobile user of tomorrow to live in a world full of digital services and artifacts. Those are embedded into a larger pervasive and ubiquitous computing environment. In this scenery of mobile and highly dynamic systems, communication and interaction plays a role of outstanding importance. It can be foreseen that in environments with a large number of independent digital services and clients many well-known paradigms for distributed systems simply fall short. For instance, tightly coupled and peer-to-peer communication is hardly maintainable. Mobile clients appear and disappear constantly or are frequently switched off for saving energy. Pervasive systems hence have to cope with a large degree of dynamics and change, which must be handled appropriately in a loosely-coupled fashion.

Moreover, the need to access different services at any time and any place forces service provision to shift to a data-centric view on interaction. The actual content of data ("what") is more important for applications than the identity of the producer of data ("who"). Consequently, interaction between producers and consumers is dominated by the active data in the system. The same data-centric view is necessary when considering the relationships between producers and consumers of data. A single data item in such large scale settings potentially is interesting for more than a single consumer. For example, the current reading of a sensor embedded in the surroundings may be used for many applications that act on the data item. On the other hand, "raw data" is not always suitable as direct input to applications. Often, on a higher level, the need for aggregation and interpretation exists. This is especially true for *context-sensitive* applications, adapting to the environment a mobile client currently is located at. Adaptation is an important criterion for such applications, as they often react to events which occur in the real world. For instance, consider the situation where a person enters a room. While this event happens in the physical world, it also is reflected as notification in the digital world. Effectively, this requires efficient many-to-many communication together with a number of operations on such data in the system.

Efficiency of notification delivery is also needed when considering the dimensions such system are supposed to grow to. Both, in physical extension, as well as computational complexity, pervasive systems are expected to grow to large scales.

The underlying idea of this thesis was to find answers to the general problem of how the proverbial billions of mobile and smart devices, services, and artifacts can be orchestrated in such highly dynamic environments. Our belief is that this can only be done with strong support from an infrastructure facilitating common requirements for pervasive systems. The approach we pursued throughout this thesis was to use a well-known and successfully deployed infrastructure for distributed systems and evolve it gradually into a new middleware, thereby integrating new concepts and paradigms into its core functionality. A striking candidate for such an infrastructure to use as a basis is the publish/subscribe paradigm. It already supports important facets of mobile and pervasive environments,

like loose coupling and scalability. But, on the other hand, the publish/subscribe paradigm is optimized for rather static distributed systems and therefore in many important aspects falls short for the intended use in mobile and pervasive systems.

10.1 Results

This thesis shows in general how a distributed notification service can be leveraged to support pervasive and ubiquitous computing environments. Starting with a detailed analysis of the requirements for such support in Chapter 2, we clearly stated where common middleware, as successfully deployed for static distributed systems, falls short under the conditions found in pervasive systems. Additionally, we identified the key requirements necessary for the support of mobile clients. Of outstanding importance among those is the proper support for mobility in such settings. Hence, Chapters 5, 6, and 7, are concerned with the implications of supporting mobility with a distributed event broker network.

In Chapter 5, the main result is a solution for the *transparent* support of mobility. Transparency of mobility is a common requirement for mobile clients that roam freely. Certain aspects of mobility, like the change of broker the client is attached to, are handled transparently for the client concerned. This transparency is important for the evolution of the publish/subscribe paradigm from a static system setting towards its use in mobile and hence highly dynamic settings. A relocation algorithm is presented that facilitates location transparency, offering the possibility to transfer existing event-based applications to mobile scenarios as well as supporting mobility-aware applications. The solution is based on the detailed analysis of the requirements for such support and clearly shows its impact on the underlying publish/subscribe notification service. The algorithm seamlessly extends the existing content-based routing infrastructure, the REBECA notification service, to support non-interrupted, sender-FIFO ordered delivery of notifications to moving clients, which need not to be aware of this extension. This is of particular importance for the use together with legacy applications. Moreover, no central repository or control nor any communication outside of the publish/subscribe infrastructure is needed. On the other hand, applications can still benefit from the service's inherent mechanisms, like advanced routing algorithms. The presented solution for mobile clients in publish/subscribe systems transfers the characteristics of the publish/subscribe paradigm to mobile scenarios in an appropriate way. Loose coupling and drawing from notification delivery localities is explicitly supported.

Chapter 6 advances support of mobility in the infrastructure an additional step further and introduces location-dependent subscriptions and notifications. Together with the foundations of location models and the reference location model introduced in Chapter 4, the publish/subscribe infrastructure can cater to location-sensitivity of mobility-aware mobile clients. The main challenge here is twofold: first, the details of adapting location-dependent subscriptions after location changes have to be handled in the infrastructure, as opposed to a naïve implementation, where the client handles them itself; and second, efficient, non-interrupted notification delivery can degenerate to network flooding without proper countermeasures in the publish/subscribe notification service. Network flooding is costly and easily can render a small, resource-limited device useless, due to the need for client-side filtering. We introduced an adaptive algorithmic solution which addresses the challenges named above. Furthermore, mobile clients can make use of a specification language for stating their interest for location-dependent information. Location-dependent subscriptions subsequently are evaluated and adapted by the broker network. Additionally, an explicit placeholder (`myLoc`) is introduced,

stating the interest in some information relating to the current position. Adaptation of this placeholder is done in the infrastructure. Moreover, by using location-dependent subscriptions, the client defines the range of relevant information for its thread of execution. This is leveraged for constraining the degree to which *uncertainty* of location-related message delivery is necessary to allow for an uninterrupted and timely delivery without resorting to network flooding. Thereby, an effective means to express and implement location-awareness is introduced.

However, neither solution can prevent that a client needs a certain initialization phase to properly adapt to location or context changes. Mobility support, as introduced in Chapter 5, decouples producer and consumers of non-context-dependent notifications in space and time. Chapter 6, on the other hand, minimizes the bootstrapping latency of clients depending on location-dependent data within the boundaries of a single broker. The solution presented in Chapter 7 combines the characteristics of both approaches for roaming clients. In general, only after some sequence of events an application commences its normal operation from a valid state. Due to the asynchronous nature of event-driven systems, this can severely impair the usability of the publish/subscribe paradigm in pervasive systems. Hence, we devised techniques in the infrastructure, enabling a mobility-aware client to access notifications already delivered in the past. The main problem we addressed is the minimization of the time-span a client has to "listen" to a concrete stream of event notifications before it is able to resume operation. This latency time can become a major problem in scenarios where clients are roaming. As an example we employed a common scenario where a roaming client misses some crucial information by the proverbial "fraction of a second", leaving an application at a new location in an inconsistent state. To address this problem, we established additional buffers in the broker network of the distributed notification service. Additionally, we devised a set of search and consolidation strategies tailored to minimize the bootstrapping latency experienced by a client attached to the broker network. Thereby, a considerable and important decoupling of producers and consumers in space *and* time is achieved.

One of the central results of Chapter 2 is the identification of *context* as a necessary source of information for the operation and adaptation of a context-aware application, as well as an important factor for the optimization of message delivery in the infrastructure. In general, applications that make use of external information in order to react to changes in a volatile execution environment are reactive in nature. This reactive and adaptive behavior has a serious impact on the programming model as well as the data model an application has to cope with. In Chapter 8 we introduced a model for the structured development of such context-sensitive applications. We leveraged the notion of control-based coordination as a convenient abstraction for the specification of applications based on finite state machines. Then, we showed how this specification can be mapped to a proper set of subscription operating on data available in the current execution environment. The main challenge we faced was the heterogeneity of the data sources common in pervasive computing systems. We introduced a formal framework for the structured mapping of a task-oriented and control-driven application specifications to data items and sources available in the system. This included the use of certain operations on actual data items, such as aggregation, composition, and interpretation of notifications.

Summing up, this thesis presents several important and novel extensions for evolving the publish/subscribe paradigm into an efficient and convenient means for communication and interaction in pervasive computing environments. We analyzed the key factors where pervasive computing environments fundamentally are different from conventional distributed systems. Subsequently, we showed the general appropriateness of the publish/subscribe paradigm for facilitating some of the basic requirements in highly dynamic and mobile systems. But in several aspects publish/subscribe

falls short. There, we introduced necessary extensions which address the shortcomings of publish/subscribe, like support for mobile clients or location- and context-awareness. A distributed publish/subscribe notification service with the extensions presented in this thesis constitutes an appropriate platform for building pervasive and ubiquitous computing systems.

10.2 Future Work

Obviously, a number of issues have not been discussed further in this thesis and should be the subject of further research.

In this thesis we have taken a rather opportunistic approach for the decoupling of producers and consumers in time. The general topic of caching and event histories in a distributed publish/subscribe notification service opens up a wide area of research opportunities. Especially its application to mobile systems, as we have done in this thesis, raises important and interesting questions. For instance, optimality of buffer placement in the network strongly depends on the degree of location-dependency of subscriptions and notifications and is hardly tackled.

Another important issue in pervasive computing systems is the question of security and trust. In an ever changing and potentially hostile environment, in the long term, it is of vital importance to have some means to protect clients from malicious surroundings and vice versa. This obviously includes client access control as well as securing the integrity of data in the system. However, only first ideas on securing distributed publish/subscribe notification services exist (e.g., [DKHP03; BEP+03; WCEW02; FZB+04]), leaving open a wide field for future research.

The last item we want to mention is the issue of scaling down the infrastructure for a more flexible and versatile range of possible system domains. Currently a nomadic system model is assumed. Moreover, the publish/subscribe service itself relies on rather reliable and static connections between the brokers. However, in more dynamic settings, as we envision them for example in wireless sensor networks (WSN), those assumptions are hard to maintain. There, the infrastructure itself must be scaled down and the broker network built-up flexibly and fault-tolerant. Besides first ideas in [TBF+03], this also is left for future work.

Summing up, this thesis provides numerous solutions to existing problems motivated by using the publish/subscribe paradigm in pervasive computing systems. At the same time it also raises a considerable number of questions and shows possible as well as necessary research directions for future work.

Bibliography

[AAH+97] Gregory D. Abowd, Christopher G. Atkeson, Jason Hong, Sue Long, Rob Kooper, and Mike Pinkerton. Cyberguide: a mobile context-aware tour guide. *Wireless Networks*, 3(5):421–433, 1997.

[ACG86] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *Computer*, pages 26–34, August 1986.

[ACH+01] Mike Addlesee, Rupert Curwen, Steve Hodges, Joe Newman, Pete Steggles, Andy Ward, and Andy Hopper. Implementing a sentient computing system. *IEEE Computer*, 34(8):50–56, Aug 2001.

[ADH+99] Gerd Aschemann, Svetlana Domnitcheva, Peer Hasselmeyer, Roger Kehr, and Andreas Zeidler. A framework for the integration of legacy devices into a Jini management federation. In R. Stadler and B. Stiller, editors, *Tenth IFIP/IEEE International Workshop on Distributed Systems: Operations and Management DSOM '99, Zurich, Switzerland*, volume 1700 of *Lecture Notes in Computer Science*, pages 257–268. Springer-Verlag, October 11–13, 1999.

[AF00] Mehmet Altinel and Michael J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *The VLDB Journal*, pages 53–64, 2000.

[AFZ97] Swarup Acharya, Michael Franklin, and Stanley Zdonik. Balancing push and pull for data broadcast. In *Proc. of ACM SIGMOD*, pages 183–194, May 13–15 1997.

[AKZ99] Gerd Aschemann, Roger Kehr, and Andreas Zeidler. A Jini-based Gateway Architecture for Mobile Devices. In *Proceedings of the Java-Informations-Tage 1999 (JIT99)*, pages 203–212. Springer-Verlag Berlin Heidelberg, 1999.

[AM00] Gregory D. Abowd and Elizabeth D. Mynatt. Charting past, present, and future research in ubiquitous computing. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):29–58, 2000.

[AS96] Varol Akman and Mehmet Surav. Steps towards formalizing context. *AI Magazine*, 17(3):55–72, 1996.

[Bal97] A. Ballardie. RFC 2201: Core based trees (CBT) multicast routing architecture, September 1997. Status: Experimental.

[BB02] Guruduth Banavar and Abraham Bernstein. Software infrastructure and design challenges for ubiquitous computing applications. *Communications of the ACM*, 45(12):92–96, 2002.

[BBC97] P.J. Brown, J.D. Bovey, and X. Chen. Context-aware applications: From the laboratory to the marketplace. *IEEE Personal Communications*, 4(5):58–64, 1997.

[BBG+00] Guruduth Banavar, James Beck, Eugene Gluzberg, Jonathan Munson, Jeremy Sussman, and Deborra Zukowski. Challenges: an application model for pervasive computing. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 266–274. ACM Press, 2000.

[BBHM95] Jean Bacon, John Bates, Richard Hayton, and Ken Moody. Using events to build distributed applications. In *IEEE SDNE Services in Distributed and Networked Environments*, pages 148–155, Whistler, British Columbia, June 1995.

[BBMS98] John Bates, Jean Bacon, Ken Moody, and Mark Spiteri. Using events for the scalable federation of heterogeneous components. In Paulo Guedes and Jean Bacon, editors, *Proceedings of the 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications*, Sintra, Portugal, September 1998.

[BBR02] Martin Bauer, Christian Becker, and Kurt Rothermel. Location models from the perspective of context-aware applications and mobile ad hoc networks. *Personal Ubiquitous Comput.*, 6(5-6):322–328, 2002.

[BEP+03] Andras Belokosztolszki, David M. Eyers, Peter R. Pietzuch, Jean Bacon, and Ken Moody. Role-based access control for publish/subscribe middleware architectures. In Jacobsen [Jac03].

[Ber96] Philip A. Bernstein. Middleware: a model for distributed system services. *Communications of the ACM*, 39(2):86–98, 1996.

[BFC93] Tony Ballardie, Paul Francis, and Jon Crowcroft. Core based trees (cbt). In I. Chlamtac, editor, *Conference Proceedings on Communications Architectures, Protocols and Applications (SIGCOMM'93)*, pages 85–95, San Francisco, CA, USA, 1993. ACM Press.

[BFG+02] Jean Bacon, Ludger Fiege, Rachid Guerraoui, H.-Arno Jacobsen, and Gero Mühl, editors. *1st Intl. Workshop on Distributed Event-Based Systems (DEBS'02)*, Vienna, Austria, 2002. IEEE Press. Published as part of the ICDCS '02 Workshop Proceedings.

[BGS01] Michael Beigl, Hans-Werner Gellersen, and Albrecht Schmidt. Medicups: Experience wih design and use of computer-augmented everyday objects. *Computer Networks*, 35(4):401–409, March 2001.

[BKS+99] G. Banavar, M. Kaplan, K. Shaw, R.E. Strom, D.C. Sturman, and Wei Tao. Information flow based event distribution middleware. In *19th IEEE International Conference on Distributed Computing Systems Workshops on Electronic Commerce and Web-based Applications/Middleware*, pages 114–121, 1999.

[BMB+00] Jean Bacon, Ken Moody, John Bates, Richard Hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel, and Mark Spiteri. Generic support for distributed applications. *IEEE Computer*, 33(3):68–76, 2000.

[BR02] Martin Bauer and Kurt Rothermel. Towards the observation of spatial events in distributed location-aware systems. In Roland Wagner, editor, *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops*, pages 581–582. Los Alamitos, California: IEEE Computer Society, 2002.

[Bro96] P. J. Brown. The stick-e document: a framework for creating context-aware applications. In *Proceedings of EP'96, Palo Alto*, pages 259–272. also published in *EP–odd*, January 1996.

[BZA03] Sumeer Bhola, Yuanyuan Zhao, and Joshua Auerbach. Scalably supporting durable subscriptions in a publish/subscribe system. In *Proceedings of 2003 International Conference on Dependable Systems and Networks (DSN'03), June 22 - 25, 2003, San Francisco, California*, pages 57–66, 2003.

[CBB03] Mariano Cilia, Christof Bornhoevd, and Alejandro P. Buchmann. CREAM: An infrastructure for distributed, heterogeneous event-based applications. In *Proceedings of the Intl Conference on on Cooperative Information Systems (CoopIS'03) (to appear)*, Nov 2003.

[CCW03] Mauro Caporuscio, Antonio Carzaniga, and Alexander L. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. Technical Report CU-CS-944-03, Department of Computer Science, University of Colorado, January 2003.

[CD85] David R. Cheriton and Stephen E. Deering. Host groups: a multicast extension for datagram internetworks. In William P. Lidinsky and Bart W. Stuck, editors, *Proceedings of the Ninth Symposium on Data Communications*, pages 172–179, Whistler Moutain, British Columbia, Canada, 1985. ACM Press.

[CD01] Gianpaolo Cugola and Elisabeth Di Nitto. Using a publish/subscribe middleware to support mobile computing. In *Proceedings of the Workshop on Middleware for Mobile Computing*, Heidelberg, Germany, November 2001.

[CDF01] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9), 2001.

[CEM01] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. Reflective middleware solutions for context-aware applications. *Lecture Notes in Computer Science*, 2192:126–132, 2001.

[CFBS97] Jeremy R. Cooperstock, Sidney S. Fels, William Buxton, and Kenneth C. Smith. Reactive environments: Throwing away your keyboard and mouse. *Communications of the ACM*, 40(9):65–73, 1997.

[CFH+03] Mariano Cilia, Ludger Fiege, Christian Haul, Andreas Zeidler, and Alejandro Buchmann. Looking into the past: Enhancing mobile publish/subscribe middleware. In Jacobsen [Jac03].

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communication of the ACM*, 32(4):444–458, April 1989.

[Cil02] M. Cilia. *An Active Functionality Service for Open Distributed Heterogeneous Environments*. Ph.D. Thesis, Department of Computer Science, Technische Universität Darmstadt, Darmstadt, Germany, August 2002.

[CIP02] Mauro Caporuscio, Paola Inverardi, and Patrizio Pelliccione. Formal analysis of clients mobility in the Siena publish/subscribe middleware. Technical report, Department of Computer Science, University of L'Aquila, October 2002.

[CK00] Guanling Chen and David Kotz. A survey of context-aware mobile computing research. Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College, November 2000.

[CK01] Guanling Chen and David Kotz. Supporting Adaptive Ubiquitous Applications with the SOLAR System. Technical Report TR2001-397, Hanover, NH, 2001.

[CK02a] Guanling Chen and David Kotz. Context aggregation and dissemination in ubiquitous computing systems. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, pages 105–114. IEEE Computer Society Press, June 2002.

[CK02b] Guanling Chen and David Kotz. Solar: An open platform for context-aware mobile applications. In *Proceedings of the First International Conference on Pervasive Computing (Short paper)*, pages 41–47, June 2002. In an informal companion volume of short papers.

[CMPC03] Paolo Costa, Matteo Migliavacca, Gian Pietro Picco, and Gianpaolo Cugola. Introducing reliability in content-based publish-subscribe through epidemic algorithms. In Jacobsen [Jac03].

[CNP00] G. Cugola, E. D. Nitto, and G. P. Picco. Content-based dispatching in a mobile environment. In *In Workshop su Sistemi Distribuiti: Algorithmi, Architectture e Linguaggi (WSDAAL)*, 2000.

[Con03] Things That Think Consortium. Homepage. Available at *http://ttt.media.mit.edu/*, 2003.

[Cor03] IBM Corp. Websphere MQ (mqseries), 2003. *http://www.software.ibm.com/mqseries*.

[CRW99] Antonio Carzaniga, David R. Rosenblum, and Alexander L. Wolf. Challenges for distributed event services: Scalability vs. expressiveness. In W. Emmerich and V. Gruhn, editors, *ICSE '99 Workshop on Engineering Distributed Objects (EDO '99)*, May 1999.

[CRW00] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC-00)*, pages 219–228, NY, July 16–19 2000. ACM Press.

[CRW01] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[CTB+95] J.R. Copperstock, K. Tanikoshi, G. Beirne, T. Narine, and W. Buxton. Evolution of a reactive environment. In *Proceedings of CHI'95*, pages 170 – 177, 1995.

[DA00] Anind K. Dey and Gregory D. Abowd. Towards a better understanding of context and context-awareness. In *Proceedings of the Workshop on What, Who, Where, When and How of Context-Awareness, affiliated with the CHI2000 Conference on Human Factors in Computer Systems*. New York, NY: ACM Press, 2000.

[DC90] Stephen E. Deering and David R. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990.

[DCEF02] Nigel Davies, Keith Cheverst, Christos Efstratiou, and Adrian Friday. The rationale for infrastructure support for adaptive and context-aware applications: A position paper. In Birgitta König-Ries, Kia Makki, S. A. M. Makki, Niki Pissinou, and Peter Scheuermann, editors, *Infrastructure for Mobile and Wireless Systems*, volume 2538 of *Lecture Notes in Computer Science*, pages 146–152. Springer, 2002.

[DCME01] Nigel Davies, Keith Cheverst, Keith Mitchell, and Alon Efrat. Using and determining location in a context-sensitive tour guide. *IEEE Computer*, 34(8):35–41, 2001.

[Dee89] Stephen Deering. Host Extensions for IP Multicasting. Request for Comments (RFC) 1112, August 1989.

[DFWB98a] Nigel Davies, Adrian Friday, Stephen P. Wade, and Gordon S. Blair. An asynchronous distributed systems platform for heterogeneous environments. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 66–73. ACM Press, 1998.

[DFWB98b] Nigel Davies, Adrian Friday, Stephen P. Wade, and Gordon S. Blair. L2imbo: a distributed systems platform for mobile computing. *Mobile Networks and Applications (MONET). Special issue on protocols and software paradigms of mobile networks.*, 3(2):143–156, 1998.

[DKHP03] Boris Dragovic, Evangelos Kotsovinos, Steven Hand, and Peter R. Pietzuch. Xenotrust: Event-based distributed trust management. In *Proceedings of the 14th International Workshop on Database and Expert Systems Applications*, 2003.

[DMDP03] Sergio Duarte, J. Legatheaux Martins, Henrique J. Domingos, and Nuno Preguica. A case study on event dissemination in an active overlay network environment. In Jacobsen [Jac03].

[DR03] Frank Dürr and Kurt Rothermel. On a location model for fine-grained geocast. In Anind K. Dey, Albrecht Schmidt, and Joseph F. McCarthy, editors, *UbiComp 2003: Ubiquitous Computing, 5th International Conference, Seattle, WA, USA, October 12-15, 2003, Proceedings*, volume 2864 of *Lecture Notes in Computer Science*. Springer, 2003.

[DRD+00] Alan Dix, Tom Rodden, Nigel Davies, Jonathan Trevor, Adrian Friday, and Kevin Palfreyman. Exploiting space and location as a design framework for interactive mobile systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(3):285–321, 2000.

[DSA01] Anind K. Dey, Daniel Salber, and Gregory D. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Special issue on context-aware computing in the Human-Computer Interaction (HCI) Journal*, 16(2-4):97–166, 2001.

[DSAR03] Michael C. Daconta, Kevin T. Smith, Donald Avondolio, and W. Clay Richardson. *More Java Pitfalls*. Wiley Publishing, Inc., Indianapolis, Indiana, 2003.

[Edw99] W. Keith Edwards. *Core JINI*. The Sun Microsystem Press - Java Series. Pentice Hall PTR, 1999.

[EF91] Max Egenhofer and Robert Franzosa. Point-set topological spatial relations. *International Journal of Geographical Information Systems*, 5(2):161–174, 1991.

[EGD01] Patrick Th. Eugster, Rachid Guerraoui, and Christian Heide Damm. On objects and events. In Linda Northrop and John Vlissides, editors, *Proceedings of the OOPSLA '01 Conference on Object Oriented Programming Systems Languages and Applications*, pages 254–269, Tampa Bay, FL, USA, 2001. ACM Press.

[EHAB99] Mike Esler, Jeffrey Hightower, Tom Anderson, and Gaetano Borriello. Next century challenges: Data-centric networking for invisible computing. In *Mobile Computing and Networking*, pages 256–262, 1999.

[Emm00] Wolfgang Emmerich. Software engineering and middleware: a roadmap. In *Proceedings of the conference on The future of Software engineering (ICSE 2000) - Future of SE Track*, pages 117–129, Limerick, Ireland, 2000. ACM Press.

[Env03] K Desktop Environment. Kweather applet. Available at *http://www.kde.org/*, 2003.

[FGHZ03] Ludger Fiege, Felix C. Gärtner, Sidath B. Handurukande, and Andreas Zeidler. Dealing with uncertainty in mobile publish/subscribe middleware. In 1st International Workshop on Middleware for Pervasive and Ad-Hoc Computing, 2003.

[FGKZ02] Ludger Fiege, Felix C. Gärtner, Oliver Kasten, and Andreas Zeidler. Supporting mobility in content-based publish/subscribe middleware. Technical Report IC/2003/11, Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, Lausanne, Switzerland, March 2002.

[FGKZ03] Ludger Fiege, Felix C. Gärtner, Oliver Kasten, and Andreas Zeidler. Supporting mobility in content-based publish/subscribe middleware. In *IFIP/ACM Middleware 2003*, 2003.

[FJ98] D. Franklin and Flaschbart J. All gadget and no representation makes jack a dull environment. In *AAAI 1998 Spring Symposium on Intelligent Environments*, pages 155–160. Menlo Park, CA:AAAI Press, 1998.

[FJHW00] Armando Fox, Brad Johanson, Pat Hanrahan, and Terry Winograd. Integrating information appliances into an interactive workspace. *IEEE Computer Graphics and Applications*, 20(3):54–65, 2000.

[FM00] Ludger Fiege and Gero Mühl. Rebeca Event-Based Electronic Commerce Architecture, 2000. http://www.gkec.informatik.tu-darmstadt.de/rebeca.

[FMB01] Ludger Fiege, Gero Mühl, and Alejandro Buchmann. An architectural framework for electronic commerce applications. In *Informatik 2001: Annual Conference of the German Computer Society*, 2001.

[FMG03] Ludger Fiege, Gero Mühl, and Felix C. Gärtner. Modular event-based systems. *The Knowledge Engineering Review*, 17(4):359–388, 2003.

[Fra00] Michael J. Franklin. Challenges in ubiquitous data management. *Informatics: 10 Years Back, 10 Years Ahead*, 2000/2001, 2000.

[FZB+04] Ludger Fiege, Andreas Zeidler, Alejandro Buchmann, Roger Kilian-Kehr, and Gero Mühl. Security aspects in publish/subscribe systems. In *Third Intl. Workshop on Distributed Event-based Systems (DEBS'04)*, May 2004.

[GB03] Robert Grimm and Brian Bershad. System support for pervasive applications. In A. Schiper, A.A. Shvartsman, H. Weatherspoon, and B.Y. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *LNCS*, Bertinoro, Italy, 2003. Springer-Verlag. Revised papers of the FuDiCo 2002 Workshop.

[GDH+01] R. Grimm, J. Davis, B. Hendrickson, E. Lemar, A. Macbeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, and D. Weth. Systems directions for pervasive computing. In *8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), Elmau/Oberbayern, Germany*, 2001.

[GDL+01] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, S. Gribble, T. Anderson, B. Bershad, G. Borriello, and D. Wetherall. System-level programming abstractions for ubiquitous computing. 2001.

[Gei01] Kurt Geihs. Middleware challenges ahead. *IEEE Computer*, 34(6):24–31, June 2001.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, USA, 1995.

[Gri02] Robert Grimm. *System Support for Pervasive Applications*. PhD thesis, University of Washington, Dec 2002.

[Gro97] The Open Group. *DCE 1.1: Remote Procedure Call*. Technical Standard C706. The Open Group, Cambridge, MA, USA, 1997.

[Gue04] Pablo Guerrero. Looking into the past: Enhancing mobile publish/subscribe middleware. Master's thesis, Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, Argentinia, 2004.

[HB01a] J. Hightower and G. Borriello. A survey and taxonomy of location systems for ubiquitous computing, 2001.

[HB01b] Jeffrey Hightower and Gaetano Borriello. Location systems for ubiquitous computing. *IEEE Computer*, 34(8):57–66, August 2001.

[HBBM96] Richard Hayton, Jean Bacon, John Bates, and Ken Moody. Using events to build large scale distributed applications. In *Proceedings of the ACM SIGOPS European Workshop*, pages 9–16, 1996.

[hCRSZ02] Yang hua Chu, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang. A case for end system multicast. *IEEE Selected Areas in Communications*, 20(8), 2002.

[HG97] David Harel and Eran Gery. Executable object modeling with statecharts. *Computer*, 30(7):31–42, 1997.

[HGM01] Yongqiang Huang and Hector Garcia-Molina. Publish/subscribe in a mobile environment. In *Proc. of MobiDE01*, May 2001.

[HHS+99] Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, and Paul Webster. The anatomy of a context-aware application. In *The Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM'99)*, pages 59–68, 1999.

[HHS+02] Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, and Paul Webster. The anatomy of a context-aware application. *Wireless Networks*, 8(2/3):187–197, 2002.

[HKL+99a] Fritz Hohl, Uwe Kubach, Alexander Leonhardi, Kart Rothermel, and Markus Schwehm. Next century challenges: Nexus–an open global infrastructure for spatial-aware applications. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 249–255. ACM Press, 1999.

[HKL+99b] Fritz Hohl, Uwe Kubach, Alexander Leonhardi, Kurt Rothermel, and Markus Schwehm. Nexus - an open global infrastructure for spatial-aware applications. Technical Report TR-1999-02, 25, 1999.

[HL01] J.I. Hong and J.A. Landay. An infrastructure approach to context-aware computing. *Human-Computer Interaction (HCI) Journal*, 16(2-3), 2001.

[HMNS01] Uwe Hansmann, Lothar Merk, Martin S. Nicklous, and Thomas Stober. *Pervasive Computing Handbook*. Springer-Verlag Berlin Heidelberg, 2001.

[HN96] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[HNBR97] Richard Hull, Philip Neaves, and James Bedford-Roberts. Toward situated computing. In *1st International Symposium on Wearable Computers (ISWC '97)*, pages 146–155, 1997.

[Hol04] Definition of Holistic. *http://www.pbs.org/faithandreason/gengloss/holist-body.html*, 2004.

[Huh01] Christopher Huhn. Ein benutzerfreundlicher Namensdienst für kontextsensitive Umgebungen (german). Master's thesis, Technische Universität Darmstadt, 2001.

[IBM01a] IBM. Gryphon: Publish/subscribe over public networks. Technical report, IBM T. J. Watson Research Center, 2001.

[IBM01b] IBM. IBM Pervasive Computing Homepage. Available at *http://www.ibm.com/pvc/*, 2001.

[IGE+03] Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, John Heidemann, and Fabio Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking (TON)*, 11(1):2–16, 2003.

[Inc01] Sun Microsystems Inc. Java naming and directory interface ^{TM} (jndi). *http://java.sun.com/products/jndi/*, 2001.

[Jac03] H.-Arno Jacobsen, editor. *2nd Intl. Workshop on Distributed Event-Based Systems (DEBS'03)*. ACM Press, June 2003.

[Joh95] D.B. Johnson. Scalable support for transparent mobile host internetworking. *Wireless Networks*, 1:311–321, October 1995.

[JS02] Changhao Jiang and Peter Steenkiste. A hybrid location model with a computable location identifier for ubiquitous computing. In L.E. Holmquist G. Borriello, editor, *UbiComp 2002: Ubiquitous Computing: 4th International Conference, Göteborg, Sweden, September 29 - October 1, 2002. Proceedings*, number 2498 in LNCS, pages 246–263. Springer-Verlag Heidelberg, 2002.

[KB] Tim Kindberg and John J. Barton. A web-based nomadic computing system. Whitepaper available at *http://www.cooltown.com/dev/wpapers/index.asp*.

[KB01] Tim Kindberg and John Barton. A web-based nomadic computing system. *Computer Networks, Elsevier*, 35(4), March 2001.

[KBM+] Tim Kindberg, John J. Barton, Jeff Morgan, Gene Becker, Debbie Caswell, Philippe Debaty, Gita Gopal, Marcos Frid, Venky Krishan, Howard Morris, John Schettino, and Bill Serra. People, places, things: Web presence for the real world. Whitepaper available at *http://www.cooltown.com/dev/wpapers/index.asp*.

[KBM+02] Tim Kindberg, John Barton, Jeff Morgan, Gene Becker, Debbie Caswell, Philippe Debaty, Gita Gopal, Marcos Frid, Venky Krishnan, Howard Morris, John Schettino, Bill Serra, and Mirjana Spasojevic. People, places, things: web presence for the real world. *Mobile Network Applications*, 7(5):365–376, 2002.

[KCBC02] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, 2002.

[KCM+00] Fabio Kon, Roy Campbell, M. Dennis Mickunas, Klara Nahrstedt, , and Francisco J. Ballesteros. 2K: A distributed operating system for dynamic heterogeneous environments. In *9th IEEE International Symposium on High Performance Distributed Computing. Pittsburgh. August 1-4*, 2000.

[KEG93] Wolfgang Kainz, Max Egenhofer, and Ian Greasley. Modeling spatial relations and operations with partially ordered sets. *International Journal of Geographical Information Systems*, 7(3):215–229, 1993.

[KF02] Tim Kindberg and Armando Fox. System software for ubiquitous computing. *IEEE Pervasive Computing*, 1(1):70–81, 2002.

[KRL+00] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 121–143, New York, April 2000. Springer-Verlag.

[KSC+98] Fabio Kon, Ashish Singhai, Roy H. Campbell, Dulcineia Carvalho, Robert Moore, and Francisco J. Ballesteros. 2k: A reflective, component-based operating system for rapidly changing environments. In *ECOOP Workshops*, pages 388–389, 1998.

[KVZ99] Roger Kehr, Harald Vogt, and Andreas Zeidler. Towards a generic proxy execution service for small devices. In *Proceedings of Workshop on Future Services for Networked Devices (FuSeNetD'99), Heidelberg*, November 8–9, 1999.

[Lab03] Hewlett Packard Research Labs. cooltown homepage. Available at *http://www.cooltown.com/*, 2003.

[LCB99] C. Liebig, M. Cilia, and A. Buchmann. Event composition in time-dependent distributed systems. In *Proceedings 4th IFCIS Conference on Cooperative Information Systems (CoopIS'99)*, pages 70–78, Edinburgh, Scotland, September 1999. IEEE Computer Society Press.

[Leo98] Ulf Leonhardt. *Supporting Location-Awareness in Open Distributed Systems*. PhD thesis, Dept. of Computing, Imperial College London, 1998.

[LL90] Leslie Lamport and Nancy Lynch. Distributed computing: Models and methods. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 1157–1199. Elsevier, 1990.

[LS90] Eliezer Levy and Abraham Silberschatz. Distributed file systems: concepts and examples. *ACM Comput. Surv.*, 22(4):321–374, 1990.

[LS00] Henry Lieberman and Ted Selker. Out of context: Computer systems that adapt to, and learn from , context. *IBM Systems Journal*, 39(3–4):617–632, 2000.

[Luc02] David Luckham. *The Power of Events*. Addison-Wesley, 2002.

[LXZL02] Dik Lun Lee, Jianliang Xu, Baihua Zheng, and Wang-Chien Lee. Data management in location-dependent information services. *Pervasive Computing, IEEE*, 1(3):65–72, 2002.

[Mac04] Carsten Mackenroth. Mobilitäsunterstützung in Publish/Subscribe-systemen. Master's thesis, Datenbanken und Verteilte Systeme, Technische Universität Darmstadt, 2004.

[Mat89] Friedemann Mattern. *Verteilte Basisalgorithmen*. Number 226 in Informatik Fachberichte. Springer-Verlag Heidelberg, 1989.

[Mat01] Friedemann Mattern. Ubiquitous Computing – Der Trend zur Informatisierung und Vernetzung aller Dinge (in German). In *Der Weg in die mobile Informationsgesellschaft, Tagungsband 6. Deutscher Internet-Kongress*. dpunkt-Verlag, September 2001. Available at *http://www.inf.ethz.ch/vs/publ/*.

[MC02] R. Meier and V. Cahill. STEAM: Event-based middleware for wireless ad hoc networks. In *Proc. of DEBS'02*, 2002.

[MCE02] Cecilia Mascolo, Licia Capra, and Wolfgang Emmerich. Mobile computing middleware. In E. Gregori, G. Anastasi, and S. Basagni, editors, *Advanced Lectures on Networking, NETWORKING 2002 Tutorials*, volume 2497 of *LNCS*, pages 20–58, Pisa, Italy, 2002. Springer-Verlag.

[MDA+] Michael C. Mozer, Robert H. Dodier, Marc Anderson, Lucky Vidmar, Robert P III Cruickshank, and Debra Miller. The neural network house: An overview.

[ME04] C. Maihöfer and Reinhold Eberhardt. Time-stable geocast for ad hoc networks and its application with virtual warning signs. *Special issue of Computer Communications*, to appear 2004.

[MF01] Gero Mühl and Ludger Fiege. Supporting covering and merging in content-based publish/subscribe systems: Beyond name/value pairs. *IEEE Distributed Systems Online (DSOnline)*, 2(7), 2001.

[MFB02] Gero Mühl, Ludger Fiege, and Alejandro P. Buchmann. Filter similarities in content-based publish/subscribe systems. In H. Schmeck, T. Ungerer, and L. Wolf, editors, *International Conference on Architecture of Computing Systems (ARCS)*, volume 2299 of *Lecture Notes in Computer Science*, pages 224–238, Karlsruhe, Germany, 2002. Springer-Verlag.

[MFE03] C. Maihöfer, Walter Franz, and Reinhold Eberhardt. Stored geocast. In *Proceedings of Kommunikation in Verteilten Systemen (KiVS)*, pages 257–268, Leipzig, Germany, February 2003. Springer-Verlag Heidelberg.

[MFGB02] Gero Mühl, Ludger Fiege, Felix C. Gärtner, and Alejandro P. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe systems. In *Proc. MASCOTS 2002*, 2002.

[Mic97] Sun Microsystems. *Java RMI specification*. Mountain View, CA, USA, 1997.

[MKCC01] René Meier, Marc-Olivier Killijian, Raymond Cunningham, and Vinny Cahill. Towards proximity group communication. In Banavar:2001:MobileMiddleware, editor, *Advanced Topic Workshop Middleware for Mobile Computing (Middleware 2001)*, 2001.

[Moz98] Micheal C. Mozer. The neural network house: An environment that adapts to its inhabitants. In M. Coen, editor, *Proceedings of the American Association for Artificial Intelligence Spring Symposium on Intelligent Environments*, pages 110–114. Menlo Park, CA: AAAI Press, 1998.

[MPR01] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. LIME: A Middleware for Physical and Logical Mobility. In *Proc. of ICDCS-21*, pages 524–533, May 2001.

[Müh01] Gero Mühl. Generic constraints for content-based publish/subscribe systems. In *Proc. of CoopIS '01*, volume 2172 of *LNCS*, pages 211–225. Springer-Verlag, 2001.

[Müh02] Gero Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Darmstadt Univ. of Technology, 2002.

[MW03] Merriam-Webster Online Dictionary. *http://www.m-w.com/cgi-bin/dictionary*, 2004.

[Nel98] Giles John Nelson. *Context-Aware and Location Systems*. PhD thesis, University of Cambridge, 1998.

[NI97] Julio C. Navas and Tomasz Imielinski. GeoCast – geographic addressing and routing. In *ACM/IEEE Int. Conf. on Mobile Computing and Networking (MOBICOM'97)*, pages 66–76, 1997.

[NM01] Daniela Nicklas and Bernhard Mitschang. The nexus augmented world model: An extensible approach for mobile, spatially-aware applications. In Yingxu Wang, Shushma Patel, and Ronald Johnston, editors, *Proceedings of the 7th International Conference on Object-Oriented Information Systems : OOIS '01 ; Calgary, Canada*, pages 392–401, August 2001.

[Obj00] Object Management Group. CORBA event service specification, version 1.0. OMG Document formal/2000-06-15, 2000.

[Obj02] Object Management Group. Corba notification service. OMG Document formal/2002-08-04, 2002.

[OPSS93] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The information bus—an architecture for extensible distributed systems. In Barbara Liskov, editor, *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 58–68, Asheville, NC, USA, December 1993. ACM Press.

[PA98] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. In M. Zelkowitz, editor, *The Engineering of Large Systems*, volume 46 of *Advances in Computers*. Academic Press, August 1998.

[Pag84] H. Pagels, editor. *Computer Culture: the Scientic, Intellectual and Social Impact of the Computer*, volume Volume 426 of Annals of The New York Academy of Sciences, chapter John McCarthy: Some expert systems need common sense, pages 129–137. The New York Academy of Sciences, 1984.

[PB02] Peter Pietzuch and Jean Bacon. Hermes: A distributed event-based middleware architecture. In Bacon et al. [BFG+02]. Published as part of the ICDCS '02 Workshop Proceedings.

[PCB00] Nissanka B. Priyantha, Anit Chakraborty, and Hari Balakrishnan. The cricket location-support system. In *Proceedings of the sixth annual international conference on Mobile computing and networking (MOBICOM 2000)*, pages 32–43. ACM, 2000.

[Per98a] Charles E. Perkins. Mobile networking in the Internet. *Mobile Networks and Applications*, 3(4):319–334, 1998.

[Per98b] Charles E. Perkins. Mobile Networking through Mobile IP. *IEEE Internet Computing*, 2(1):58–69, 1998.

[Pie04] Peter Robert Pietzuch. *Hermes: A Scalable Event-Based Middleware*. PhD thesis, Queens' College, University of Cambridge, February 2004.

[PLM02] George Papadopoulos, Theophilos Limniotes, and Costas Mourlas. Event-driven coordination of real-time components. In Bacon et al. [BFG+02]. Published as part of the ICDCS '02 Workshop Proceedings.

[PRM99] Jason Pascoe, Nick Ryan, and David Morse. Issues in developing context-aware computing. In Hans-Werner Gellersen, editor, *HUC*, volume 1707 of *Lecture Notes in Computer Science*, pages 208–221. Springer, 1999.

[PSB03] Peter R. Pietzuch, Brian Shand, and Jean Bacon. A Framework for Event Composition in Distributed Systems. In *Proc. of the 4th Int. Conf. on Middleware (MW'03)*, Rio de Janeiro, Brazil, June 2003.

[RC90] G.-C. Roman and H.C. Cunningham. Mixed programming metaphors in a shared dataspace model of concurrency. *IEEE Transactions on Computer Engeneering*, 16(12):1361–1373, 1990.

[Rei02] Meik Reimer. Design and implementation of a mobile personal health monitor. Master's thesis, Datenbanken und Verteilte Systeme, Technische Universität Darmstadt, 2002.

[RFC1057] Sun Microsystems Inc. RPC: Remote Procedure Call Protocol Specification Version 2 (ONCRPC). Internet RFC 1057, June 1988.

[RFC2009] T. Imielinski and J. Navas. GPS-based addressing and routing. Internet RFC 2009, Mar 1996.

[RHC+02] Manuel Román, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. A middleware infrastructure for active spaces. *Pervasive Computing, IEEE*, 1(4):74–83, 2002.

[RHKS01] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-level multicast using content-addressable networks. In Jon Crowcroft and Marcus Hofmann, editors, *Proceedings of the Third International COST264 Workshop (NGC 2001)*, volume 2233 of *LNCS*, pages 14–29. Springer-Verlag, November 2001.

[RKM02] Kay Römer, Oliver Kasten, and Friedemann Mattern. Middleware challenges for wireless sensor networks. *ACM Mobile Computing and Communication Review*, pages 59–61, October 2002.

[SA97] W. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of the 1997 Australian UNIX Users Group*, Brisbane, Australia, September 1997.

[SAB+00] Bill Segall, David Arnold, Julian Boot, Michael Henderson, and Ted Phelps. Content based routing with elvin4. In *Proceedings AUUG2K*, Canberra, Australia, June 2000.

[SAS01] Peter Sutton, Rhys Arkins, and Bill Segall. Supporting disconnectedness – transparent information delivery for mobile and invisible computing. In *Intl Symposium on Cluster Computing and the Grid*, 2001.

[Sat01] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8(4):10–17, August 2001.

[SAW94] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, 1994.

[SBG99] Albrecht Schmidt, Michael Beigl, and Hans-W. Gellersen. There is more to context than location. *Computers and Graphics*, 23(6):893–901, 1999.

[SDA99] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *Proc. Intl Conf. Human Factors in Comp. Sys. (CHI)*, pages 434–441, 1999.

[Sho76] E. Shortliffe. Mycin: Computer-based medical consultations. Elsevier, New York, 1976.

[SLM98] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4), April 1998.

[Sun88] Sun Microsystems Inc. RPC: Remote Procedure Call Protocol Specification Version 2 (ONCRPC). Internet RFC 1057, June 1988.

[Sun99a] Sun. *Jini Architecture Specification – Revision 1.0*. Sun Microsystems Inc., January 1999.

[Sun99b] Sun. *Jini Discovery and Join Specification – Revision 1.0*. Sun Microsystems Inc., January 1999.

[Sun99c] Sun Microsystems Inc. *Jini Transaction Specification – Revision 1.0*, January 1999.

[Sun01] Sun Microsystems Inc. *Jini Technology Surrogate Architecture Specification Version 0.7*, March 2001. Available at *http://developer.jini.org/exchange/projects/surrogate/*.

[Sun02a] Sun Microsystems, Inc. Java Message Service Specification 1.1, 2002.

[Sun02b] Sun Microsystems, Inc. JavaSpaces Service Specification version 1.2.1, 2002.

[Sun03] Sun Microsystems, Inc. Java 2 Platform Enterprise Edition, Version 1.4, November 2003. http://java.sun.com/j2ee/.

[TBF+03] Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Andreas Zeidler, and Alejandro P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In Jacobsen [Jac03].

[Til02] Alexander Till. Erweiterter notifikationsdienst für nexus. Master's thesis, Universität Stuttgart, 2002.

[TP00] Cheng Lin Tan and Stephen Pink. Mobicast: a multicast scheme for wireless networks. *Mobile Networks and Applications*, 5(4):259–271, 2000.

[WB96] Mark Weiser and John Seely Brown. The coming age of calm technology. Available at *http://www.ubiq.com/hypertext/weiser/acmfuture2endnote.htm*, October 1996.

[WC02] Brad Williams and Tracy Camp. Comparison of broadcasting techniques for mobile ad hoc networks. In *Proceedings of the third ACM international symposium on Mobile ad hoc networking and computing*, pages 194–205. ACM Press, 2002.

[WCEW02] Chenxi Wang, Antonio Carzaniga, David Evans, and Alexander L. Wolf. Security issues and requirements for Internet-scale publish-subscribe systems. In *Proceedings of the Thirtyfifth Hawaii International Conference on System Sciences (HICSS-35)*, Big Island, Hawaii, January 2002.

[Wei91] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, September 1991.

[Wei93] Mark Weiser. Hot topic: Ubiquitous computing. *IEEE Computer*, 26(10):71–72, 1993.

[Wei95] Mark Weiser. Mark Weiser's homepage. Available at *http://www.ubiq.com/hypertext/weiser/*, 1995.

[WGS84] WGS84 Homepage. World Geodetic System 1984 (WGS 84) specification. Available at: *http://www.wgs84.com/*, 1984.

[WHFG92] R. Want, A. Hopper, V Falcao, and J. Gibbons. The Active Badge Location System. *ACM Transactions on Information Systems*, 10(1), 1992.

[Win01] Terry Winograd. Architectures for context. *Special issue on context-aware computing in the Human-Computer Interaction (HCI) Journal*, 16(2-4), 2001.

[WJH97] Andy Ward, Alan Jones, and Andy Hopper. A new location technique for the active office. *IEEE Personal Communications*, 4(5):42–47, 1997.

[WMLF98] Peter Wyckoff, Stephen McLaughry, Tobin Lehman, and Daniel Ford. T spaces. *IBM Systems Journal*, August 1998.

[WSA+95] R. Want, B. N. Schilit, N. I. Adams, R. Gold, K. Petersen, D. Goldberg, J. R. Ellis, and M. Weiser. An overview of the PARCTAB ubiquitous computing experiment. *IEEE Personal Communications*, 2(6):28–33, Dec 1995.

[WZ99] Ralph Wittmann and Martina Zitterbart. *Multicast – Protokolle und Anwendungen*. Heidelberg: dpunkt, Verlag für digitale Technologien, 1999.

[YGM94] T. W. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the Boolean model. *ACM Transactions on Database Systems*, 19(2):332–364, 1994.

[YK03] Stephen S. Yau and Fariaz Karim. An adaptive middleware for context-sensitive communications for real-time applications in ubiquitous computing environments. *To be published in: Real-Time Systems, The International Journal of Time-Critical Computing Systems*, 2003. Kluwer Academic Publishers, Dordrecht, The Netherlands.

[YKW+02] Stephen S. Yau, Fariaz Karim, Yu Wang, Bin Wang, and Sandeep K.S. Gupta. Reconfigurable context-sensitive middleware for pervasive computing. *IEEE Pervasive Computing*, 1(3):33–40, 2002.

[ZF03] Andreas Zeidler and Ludger Fiege. Mobility support with REBECA. In *Proc. of the ICDCS Workshop on Mobile Computing Middleware*, 2003.

Curriculum Vitae

Name	Andreas Zeidler
Geburtsdatum	10. April 1971 in Mülheim a.d. Ruhr

Schule

1977 – 1981	Grundschule in Mülheim a.d. Ruhr
1981 – 1984	Vestisches Gymnasium in Bottrop
1984 – 1990	Lahntalschule Biedenkopf mit Abschluß der allgemeinen Hochschulreife

Studium

1990 – 1998	Studium der Informatik an der TU Darmstadt
April 98	Abschluß als Diplom-Informatiker

Berufliche Tätigkeit

Sep 91 – Jun 95	Studentische Hilfskraft am Institut für Photogrammetrie&Kartographie, Fachbereich Vermessungswesen, TU Darmstadt
Jun 95 – Aug 98	Freiberuflicher Mitarbeiter der Firma Neeb&Partner GmbH, Darmstadt
Apr 98 – Aug 98	Wissenschaftliche Hilfskraft mit Abschluss in der Rechnerbetriebsgruppe (RBG), Fachbereich Informatik, TU Darmstadt
Aug 98 – Jun 99	Wissenschaftlicher Mitarbeiter in der Rechnerbetriebsgruppe und am Fachgebiet Verteilte Systeme, Fachbereich Informatik, TU Darmstadt
Jul 99 – Okt 03	Wissenschaftlicher Mitarbeiter am Fachgebiet Datenbanken und Verteilte Systeme des Fachbereichs Informatik, TU Darmstadt
seit Mai 04	Anstellung als Research Scientist bei der Firma Siemens AG, Abteilung Corporate Technology, Software&Engineering, Architecture (CT SE2)